PROYECTO FINAL DE CARRERA

Estudio de los códigos turbo como códigos fingerprinting

Autor: Joan Pérez Esteban
Director: Joan Tomàs Buliart
Departament d'Enginyeria Telemàtica.
Universitat Politécnica de Catalunya.
C/ Jordi Girona 1 y 3.
Campus Nord, Mod C3, UPC.
08034 Barcelona. Spain.

15 de octubre de 2010

"Las pequeñas construcciones pueden terminarlas sus propios arquitectos; las grandes y auténticas dejan siempre la piedra de clave a la posteridad"

Herman Melville, *Moby Dick*

Agradecimientos

En primer lugar agradecer a mi amigo y director de proyecto Joan Tomàs Buliart, por los buenos consejos dados a lo largo de la carrera, y sobre todo por su perseverancia, constancia y eterna paciencia mostrada en el transcurso de este proyecto.

A los amigos conocidos en esta carrera en especial a Javier Vilalta y David López por tan buenos momentos pasados cuando los tres comenzamos el pfc, a Javier Sorribas amigo insustituible desde el primer día de carrera, así como a Uri, Victor Moreno, Grant, Herranz, Ivan, Dani, Alex, Aram, Jorge, David, Rubén, Marta Adán, Andrea y muchos más que me dejo.

A mis amigos de Agramunt.

A Rodrigo y al grupo de SSE de Sun Microsystems un verdadero ejemplo de talento y compañerismo, de los que tanto he aprendido durante estos tres años, esperemos que los momentos difíciles actuales no sean duraderos.

Agradecimientos especiales a Marta Pintó, por sus valiosas correcciones su apoyo y su exigencia, sin los cuales no habría finalizado este proyecto.

Agradecer a mi madre Angela, mi hermano David y mi hermana Gemma, recién estrenada mamá, por la seguridad de que sin su infinita ayuda, comprensión y apoyo incondicional nada de esto hubiera tan siquiera empezado.

Finalmente quiero dedicar este proyecto y carrera a la memoria de mi padre Juan cuyos valiosos consejos y compañía echo tanto de menos.

Muchas gracias a todos.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Este proyecto se engloba dentro del proyecto de tesis del ingeniero Joan Tomàs Buliart, en el que se realiza un estudio de la protección de los derechos de autor mediante la combinación de técnicas de *watermarking* y métodos de codificación iterativa. Así, en este proyecto nos centraremos en la parte de *fingerprinting* y de la codificación iterativa mediante la codificación turbo. El núcleo de estudio será realizar una plataforma de codificación y decodificación turbo para la creación de marcas digitales que sirvan de identificación para cualquier medio que requiera la protección de su propiedad intelectual. Una vez realizado esto, se verificará la viabilidad de los códigos turbo según su robustez frente a ataques por confabulación de distintos usuarios deshonestos con objetivo de crear una marca falsa que permita distribuir ilegalmente la imagen, video, o cualquier otro tipo de medio con propiedad intelectual. Por ello, se realizará también el diseño de los métodos de confabulación así como los distintos filtros necesarios para encontrar los posibles usuarios deshonestos.

1.1. Situación actual

Como consecuencia del avance de las tecnologías de la información y la comunicación, unido al hecho de la práctica ausencia de barreras en el mundo virtual que suponen las comunicaciones digitales a través de la red, ha sido motivado la aparición de un mercado basado en la ausencia de propiedad

y la copia ilegal, un mercado donde se puede encontrar cualquier tipo de contenido de forma fácil y rentable. A través de las plataformas P2P, o las páginas de enlace podemos encontrar cualquier tipo de información sin tener que pagar por ella, muchas veces en detrimento de la calidad, la fiabilidad o el tiempo de descarga, aspectos que muchas veces pasan a segundo plano cuando tenemos lo que queremos y cuando lo queremos de forma instantánea y gratuita.

Según el *Observatorio de piratería y hábitos de consumo de contenidos digitales*, en los últimos seis meses del año 2009 los contenidos digitales pirateados en España se estimaron en un valor de 5.121 millones de euros, más de el triple de los 1.653 millones que supuso el consumo legal. En el segundo semestre del año 2009 el valor de la distribución ilegal supuso 2.382,5 millones de euros en películas, 2.291,6 millones de euros en música, 246,2 millones en videojuegos y 200,5 millones de euros en libros, caso este último que puede ir en ascenso gracias al nuevo soporte de digitalización de libros, $e-book$. Todo intento por frenar el incremento de estas prácticas fraudulentas ha sido en vano hasta el momento y los datos rebelan que no sólo disminuyen, sino que han aumentado. Se han dado a conocer organizaciones como la SGAE, muchas veces gracias a medidas impopulares como las soluciones encaminadas directamente a sufragar los gastos de la piratería aplicando impuestos a los medios de distribución por el simple hecho de ser capaces de poder reproducir y almacenar contenidos fraudulentos.

Muchas voces abogan por el abaratamiento del precio de los productos como solución al fraude digital, bajo el razonamiento de que al distribuir los contenidos por medio de la red se suprimen los costes de distribución por los métodos tradicionales. Seguramente se deberá tender a soluciones razonables como ésta. Lo que parece claro es que no va aparecer a medio plazo ninguna solución única que evite el fraude digital, y es por ello que soluciones tecnológicas de protección a *posteriori* como el *watermarking* y el *fingerprinting* pueden ser una solución para identificar y proteger los derechos de autor.

1.2. Motivación

El creciente auge de la distribución de software ilegal ha generado la necesidad de encontrar métodos digitales para la protección de la propiedad

intelectual. Dentro de estos métodos están las técnicas de marcado como el *watermarking* y el *fingerprinting*.

Como su nombre indica, el *fingerprinting*, tiene como objetivo la identificación la propiedad mediante una huella digital. Con esta finalidad, se inserta un código (huella) en partes concretas de la imagen, vídeo, u otro medio digital con el objetivo de poder identificar a su propietario si ésta es distribuida ilegalmente. De este modo, cualquier intento de falsificar el contenido privado pasa por la supresión o manipulación de la marca digital. Como se ha dicho, la huella digital se inserta en partes concretas de la imagen o *frame* de vídeo, por lo que todo aquél que pretenda alterar la secuencia de dígitos del *fingerprint* deberá primero identificar su posición exacta para no alterar la calidad de la obra. Una forma de hacer esto es mediante la comparación de dos o más obras distintas, por lo tanto con distinta huella digital. Una vez detectada la marca, una de las formas de manipulación es la de comparar las marcas obteniendo una de ellas por métodos de confabulación, como el quedarse aleatoriamente con dígitos de una u otra huella.

Considerando lo explicado, se considera necesario el estudio de soluciones eficaces que nos permitan aportar robustez a las huellas digitales frente a ataques por confabulación de usuarios fraudulentos y que en la medida de lo posible nos ayuden a detectar estos posibles usuarios a partir de la huella digital fraudulenta.

1.3. Conocimientos previos

Por la naturaleza de este proyecto, será necesario un detallado trabajo de documentación y conocimiento de distintas áreas de conocimiento relacionadas con la codificación de canal y su simulación. Así, algunos de los aspectos que más esfuerzo han supuesto son los siguientes:

- **Codificación de canal:** Como solución para la protección de marcas digitales, se utilizarán técnicas de codificación de canal. Es por ello que se deberá estar familiarizado con las características y precedentes de los distintos tipos de códigos. Así como los aspectos básicos que engloban a la mayoría de ellos como la ganancia de codificación o la distancia del código. También será necesario estar familiarizado con aspectos mas concretos de cada codificación, como puede ser el trellis,

la matriz generadora o la matriz de chequeo de paridad. Todos estos conocimientos son necesarios para tener una visión amplia de las técnicas de codificación y poder combinar distintos códigos en caso de ser necesario.

- **Codificación Turbo:** Pese a que pudiese quedar englobado en el punto anterior, es conveniente separar esta codificación en concreto. El núcleo del proyecto consiste en realizar una plataforma que utilice los códigos turbo en la generación de huellas digitales. Así, será de vital importancia dominar todos los aspectos de este tipo de códigos. Será necesario tener presente las características de estos códigos para poder analizar y comprender el comportamiento de las simulaciones y poder, de este modo, modificar las variables necesarias para realizar las pruebas de la forma lo más satisfactoria posible.

- **Programación en Matlab:** Matlab será el programa utilizado para la realización de las simulaciones y para el diseño del codificador y decodificador turbo. Así, se hace completamente necesario el estar familiarizado con este software tanto para la creación del código como para el análisis de los resultados y la generación de las gráficas.

1.4. Objetivos

Los objetivos que se deberán alcanzar una vez finalizado este proyecto son los siguientes:

- La realización de una plataforma de codificación - decodificación turbo que permita simular la robustez de estos códigos frente a ataques por confabulación.

- La creación de sistemas que simulen la confabulación de dos o más atacantes para su siguiente estudio.

- El diseño y realización de filtros que nos permitan recuperar, en la medida de lo posible, los distintos usuarios fraudulentos que han participado en la obtención de una marca.

- Ser capaces de modificar aspectos concretos de la plataforma diseñada con el objetivo de poder realizar las modificaciones pertinentes después

de haber analizado los resultados obtenidos, acercándonos cada vez más y en la medida de lo posible a los objetivos buscados.

- Estudiar de la degradación obtenida en el rendimiento de la plataforma tras la modificación de distintos factores como pueden ser el nombre de confabuladores o la longitud de las marcas.

1.5. Estructura del proyecto

El Proyecto se puede dividir en cuatro grandes bloques: el primero muestra la situación actual del fraude digital y ubica este proyecto dentro de las posibles soluciones a este tipo de fraudes; el segundo ofrece una base teórica de los principales tipos de codificación; el tercero se centra en el diseño de las soluciones propuestas y el cuarto analiza los resultados de las simulaciones realizadas. El documento se estructura en los siguientes capítulos:

- **Capítulo 1:** se introduce la problemática del fraude digital y se presentan las características de este proyecto como posible solución a la obtención de *fingerprints* falsos por medio de ataques de confabulación.
- **Capítulo 2:** se detalla el esquema de simulación propuesto en este proyecto así como las características de la naturaleza de la supresión de marcas digitales por medio de ataques por confabulación.
- **Capítulo 3:** se da una visión global de los códigos bloque y se muestran las características básicas de los códigos bloque más relevantes, así como sus distintos algoritmos de decodificación.
- **Capítulo 4:** se detallan los aspectos y características más importantes de los códigos convolucionales, el algoritmo de Viterbi y su creación mediante código Matlab como introducción a los códigos turbo.
- **Capítulo 5:** se analizan los códigos turbo y su naturaleza, se muestra el algoritmo BCJR y se muestra, mediante un ejemplo, la codificación y de codificación turbo de una secuencia de 12 bits a la que se le han introducido errores mediante el código Matlab diseñado para este proyecto.
- **Capítulo 6:** se muestran los fragmentos de código más relevantes para la realización de este proyecto.

- **Capítulo 7:** se muestran los resultados de las simulaciones realizadas.
- **Capítulo 8:** se analizan los resultados obtenidos a lo largo de la realización del proyecto así como el análisis de los resultados de las simulaciones. Se presentan, asimismo, líneas futuras de investigación.

Capítulo 2

Esquema de simulación

En este apartado nos centraremos en detallar el esquema implementado para la realización de las simulaciones. En primer lugar y con esta finalidad, se deberá familiarizar al lector con la problemática de las falsificaciones de marcas existente en el caso real. Una vez se haya definido el objetivo a analizar, se detallarán las soluciones actuales y sus limitaciones. Por último, nos centraremos en la propuesta a implementar.

2.1. Descripción de la problemática del fraude digital

Es evidente que nos hallamos inmersos en la era digital. Cualquier tipo de información, ya sean fotos, vídeos, texto, música, etc., es tratada actualmente en forma de unos y ceros. Esta forma tan simple de tratar la información, voltaje o ausencia de él, ha permitido al ser humano alcanzar metas impensables hace no mucho tiempo. Hoy en día se puede hablar por teléfono a prácticamente cualquier lugar del mundo, asistir a una reunión a miles de kilómetros por medio de videoconferencia o llevar en el bolsillo cientos de imágenes de nuestras vacaciones.

La era digital nos ha llevado a una nueva economía, una economía que se sostiene sobre los pilares de la información y el conocimiento con el objetivo de optimizar la productividad y la competencia. Esta nueva economía

tiene unas características que escapan al tradicional comercio y producción industrial, ya que el comercio y la información se gestionan a través de la red, y ésta elimina las fricciones clásicas del espacio y el tiempo como barreras para la distribución y el intercambio de bienes. A nivel comercial, el cambio también ha sido enorme, ya que mediante las nuevas tecnologías y su amplia difusión ha permitido que las grandes (y no tan grandes) compañías dispongan de grandes bases de datos donde seguro aparecemos la mayoría de nosotros, creando perfiles que después permitan canalizar las ofertas comerciales de forma óptima a fin de posicionar cada producto al usuario final. Así, si decidimos cambiarnos de compañía telefónica, seguramente recibiremos una llamada de la actual compañía haciéndonos una contra oferta para ganarse nuestra permanencia, si nos damos de alta en páginas web de música recibiremos ofertas al correo electrónico con temática musical, incluso si elegimos en un cajero bancario realizar las operaciones en cierto idioma esto quedará registrado y guardado en nuestro perfil. No es objetivo de este proyecto analizar si tales prácticas son correctas o no; lo que es indiscutible es que estamos rodeados de una cantidad apabullante de datos, datos de cuentas bancarias, datos científicos, medios audiovisuales etc. toda esta información es computada por potentes servidores, almacenada para su utilización en grandes cabinas de discos y, en última instancia, almacenada en librerías de cintas. Es obvio que tal cantidad de información necesita una regulación que evite las falsificaciones de todo tipo, así como el intrusismo en la propiedad intelectual, problemática que año tras año acarrea perdidas millonarias en lo que se denomina fraude digital.

Este fraude ha sido posible gracias al estallido de la comunicación a través de la red y la proliferación de medios de grabación de audio y vídeo cada vez más avanzados y económicos, así como la creciente capacidad de almacenamiento de datos. Otro aspecto clave en la proliferación del fraude digital ha sido la aparición de las redes p2p, una arquitectura de red distribuida que permite que miles de participantes compartan entre ellos porciones de ficheros sin necesidad de estar cordinados por ninguna instancia central, actuando cada uno de los miembros de la red como consumidores y proveedores a la vez.

2.1.1. Derechos de autor y piratería

En primer lugar, es necesario definir los conceptos de propiedad intelectual y derecho de autor.

En los términos de la Declaración Mundial sobre la Propiedad Intelectual (votada por la Comisión Asesora de las políticas de la Organización Mundial de la Propiedad Intelectual (OMPI), el 26 de junio del año 2000. Esta es entendida similarmente como "cualquier propiedad que, de común acuerdo, se considere de naturaleza intelectual y merecedora de protección, incluidas las invenciones científicas y tecnológicas, las producciones literarias o artísticas, las marcas y los identificadores, los dibujos y modelos industriales y las indicaciones geográficas."

Por otro lado, el derecho de autor se entiende como un conjunto de normas y principios que regulan los derechos morales y patrimoniales que la ley concede a los autores (los derechos de autor) por el solo hecho de llevar a cabo la creación de una obra literaria, artística, científica o didáctica, esté publicada o inédita. En derecho anglosajón se utiliza la noción de *copyright* (traducido literalmente como "derecho de copia") que, por lo general, comprende la parte patrimonial de los derechos de autor (derechos patrimoniales).

El día 3 de marzo del 2009, en el marco del Quinto Congreso Mundial sobre Lucha contra la "Piratería y la Falsificación", se concluyó que la falsificación de productos en los países integrantes del G20 generó una perdida de 100.000 millones de euros. Estas perdidas no tan sólo repercuten al propietario del software pirateado, sino también a los ingresos tributarios de los gobiernos, a la venda de software complementario, contratos de mantenimiento y, en última instancia, a millones de empleos directos e indirectos.

El estado Español registró en 2009 un índice de piratería de software del 42 %, situándose así en su nivel más alto de la historia. Las perdidas estimadas por el uso de éste sofware alcanzó los 739 millones de euros. España se sitúa, así, siete puntos por encima de la media europea. En Estados Unidos, por su parte, el nivel de piratería es del 21 %, registrándose a nivel mundial un índice de piratería del 41 %. De hecho, España consta en el Informe Especial 301, una lista elaborada cada año por la Oficina de comercio y por la Oficina de la Presidencia de EEUU.

Como es de esperar, estos conceptos de propiedad intelectual y derechos de autor no son nuevos, pues hace cientos de años que se observó la necesidad de regular la autoría de una obra mediante lo que se denomina patente. Como curiosidad, decir que la primera patente de la que se tiene constancia data del 1491 en la República de Venecia a favor de Pietro di Ravena sobre los derechos de su obra "Fénix". Pero aún podríamos ir mucho más lejos para encontrar ejemplos no quizás sobre patentes regladas, pero sí sobre la

necesidad de marcar, ocultar o validar la autoría de una obra o escrito. En el siglo V a.C., se creó la escitala para el envío de mensajes durante la guerra entre Atenas y Esparta. Este método consistía en la inclusión de símbolos innecesarios en el mensaje original. Para eliminar estos símbolos (descifrar el mensaje), se tenía que enrollar el escrito en un rodillo llamado escitala, de longitud y grosor prefijado, de modo que tan sólo los poseedores de la scitala correcta serían capaces de descifrar el mensaje. Otro ejemplo sería el Cifrado de César (siglo I a.C.), cifrado que consistía en el desplazamiento en módulo 3 del alfabeto original, de modo que para descifrar el contenido se debían desplazar módulo 3 hacia la izquierda cada letra del escrito.

El watermarking, por el contrario, no pretende cifrar la información, sino proteger la autoría de un determinado medio, ya sea foto, vídeo, audio o texto, de la copía ilegal y su posterior distribución. Para ello, el mejor procedimiento en contra de lo que se podría pensar no es ocultar la información, sino marcarla. Si bien es cierto que los los conceptos de cifrado y marcaje pueden parecer similares, no lo son para nada.

2.2. Diferencia entre criptografía y watermarking

Es de suma importancia diferenciar claramente los conceptos de cifrado y watermarking, definiendo claramente las limitaciones y características que tiene cada uno de ellos. La criptografía tiene como objetivo único ocultar la información de modo que tan sólo con la posesión de ciertas claves se pueda descifrar el mensaje (la escitala sería la clave de los ejemplos anteriores). El problema de utilizar métodos criptograficos para marcar medios audiovisuales o cualquier otro tipo de software, es que una vez descubierta la "llave", el mensaje puede ser descifrado sin problema alguno, obteniendo los mensajes originales al 100 %, privando por lo tanto de capacidad de distinción entre mensaje original y copia ilegal.

En la actualidad el watermarking se utiliza para prevenir la distribución de copias ilegales evitando así que terceras personas se lucren de la actividad ajena. En el siguiente esquema, (figura 2.1) extraído del PFC de Joan Tomás i Buliart, se muestra claramente la diferencia entre cifrado y watermarking.

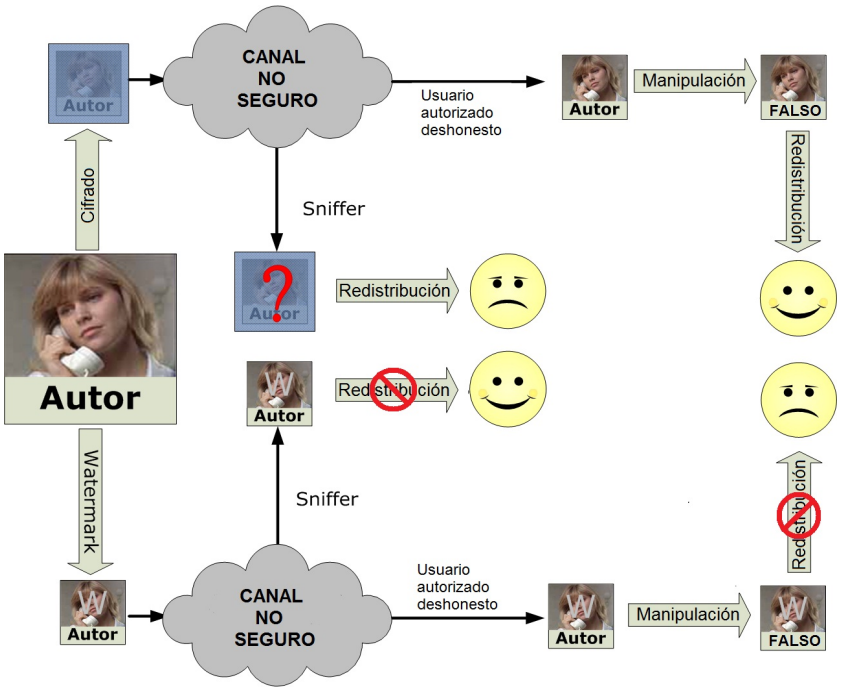


Figura 2.1: *Watermarking vs Cifrado*

Como se puede ver en la figura 2.1, el cifrado no nos solventa la problemática de la distribución ilegal de medios. Se podría pensar que la solución pasa por la creación de métodos criptográficos más efectivos y robustos, pero la experiencia dice que por muy robustos que sean los cifrados, a la larga se acaban vulnerando, y no sólo porque alguien haya descifrado la clave, sino mas bien porque algún usuario deshonesto ha distribuido su clave, dejando completamente inútiles todos los mecanismos de cifrado.

El watermarking evita que un usuario desleal comprometa el resto de copias legales, puesto que cada copia tiene una marca distinta. Claro está, que la marca debe poseer ciertas características que la hagan robusta a los ataques de usuarios deshonestos, pues una vez se haya distribuido la copia se tendrá que proteger la marca de posibles intentos para extraerla y/o manipularla.

2.3. Características de las marcas digitales

Como hemos dicho la función de una marca digital o watermark es de blindar un documento, sobre posibles manipulaciones. Por este motivo, deberá poseer las siguientes características:

- Imperceptible al ojo humano. La marca digital no debe alterar el documento original de manera sustanciosa; por ello, la marca no debería ser detectada a simple vista.
- Estadísticamente indescifrable. En un conjunto de documentos, las marcas deben ser estadísticamente lo suficientemente distantes como para que no sea posible extraer un patrón de marcaje.
- Complejas. Las marcas deben ser lo suficientemente complejas como para no ser extraídas de manera trivial por usuarios no autorizados.
- Robustas. Es necesario asegurar que los cambios de formatos, copias, emisiones, etc., no modificaran la marca, pues toda actividad legal sobre el documento debe transmitir el marcaje original.
- Resistente a manipulaciones. Es decir, la marca debe ser capaz de resistir cualquier tipo de manipulación intencionada ya sea la modificación de la mismo o el intento total de sustracción. Es en este punto en el que se ha centrado el estudio de este proyecto.
- De fácil recuperación. Esto es, la marca ha de poder extraer con una baja probabilidad de error; es decir, es necesario que la marca que hemos extraído sea la que se incrustó.

Como hemos dicho, el objetivo de este proyecto es el del análisis de los turbocódigos para crear huellas digitales que sean lo suficientemente robustas como para evitar la manipulación intencionada de la marca. Existen muchos tipos de ataque a las marcas, como pueden ser encontrar la posición de ésta y eliminarla (sustituyendo los bits de la marca por bits arbitrarios), o bien, comparando dos imágenes iguales, analizar las diferencias, extraer la marca y sacar una distinta a partir de las dos marcas originales. Es en este punto en el que queremos probar la viabilidad de los turbocódigos, no tanto para evitar la manipulación de la marcas sino, más bien, la detección de los posibles confabuladores.

Capítulo 3

Códigos bloque

Según se desprende de la teoría de codificación de canal de Shannon, un canal ruidoso por el cual una transmisión de bits se puede ver modificada, se puede, mediante sofisticadas técnicas de codificación, convertir en un canal libre de errores.

En la codificación en bloque, las secuencias de información binaria son segmentadas en bloques de mensaje de longitud fija. Cada bloque de información denotada por m consiste en k dígitos de información. Existe un total de 2^k mensajes distintos. La codificación consiste, básicamente, en asignar palabras código de longitud $n > k$ según una función biyectiva caracterizada por la adición de redundancia. Así, al conjunto de 2^k secuencias de información le corresponden 2^k palabras código, y este conjunto de 2^k palabras código es lo que se denomina *código bloque*. Hay básicamente dos modos de añadir redundancia en relación a las técnicas de control de errores: la codificación en bloque y la codificación convolucional.

En el caso de la codificación que ocupa en este capítulo, la codificación en bloque, el proceso de codificación añade (n -k) bits de redundancia, más comúnmente llamados parity check bits o bits de paridad. El proceso de codificación o sea la adición de estos bits de paridad se lleva a cabo de tal manera que siempre sea posible realizar la operación inversa para que el mensaje original m pueda ser correctamente recuperado.

Los códigos bloque se denotan por $C_b(n, k)$ y el ratio del código se denota por $R_c = k/n$ y representa una medida del nivel de redundancia del código.

3.1. Códigos Bloque lineales

Los códigos bloque lineales poseen propiedades de linealidad de modo que se reduce notablemente el proceso de codificación.

Definición: *Un código bloque de longitud n y 2^k palabras código se denomina lineal (n,k) si, y sólo si hay 2^k palabras código de un subespacio vectorial k-dimensional del espacio de todas las n-uplas de un campo de galois GF(2).*

Se puede demostrar que un código es lineal si, y sólo si, de la suma módulo 2 de dos palabras código se genera otra palabra código.

3.1.1. Matriz generadora G

Como se ha comentado anteriormente, un código bloque C(n,k) es un subespacio vectorial del espacio vectorial V_n, donde encontraremos k vectores linealmente independientes correspondientes a las palabras código $g_0, g_1, \ldots g_{k-1}$, cada una de las cuales se puede obtener de la combinación lineal de las otras:

$$c = m_0 g_0 + m_1 g_1 + \ldots + m_{k-1} g_{k-1} \tag{3.1}$$

Colocando estas palabras código linealmente independientes como columnas de una matriz k x n, obtenemos la matriz generadora G:

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{bmatrix} = \begin{bmatrix} g_{00} & g_{01} & \cdots & g_{0,n-1} \\ g_{10} & g_{11} & \cdots & g_{1,n-1} \\ \vdots & \vdots & & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} \tag{3.2}$$

De este modo la generación de palabras código se obtiene de la multiplicación entre el vector correspondiente al mensaje de información y la matriz generadora:

$$\boldsymbol{c} = \boldsymbol{m} \circ \boldsymbol{G} = (m_0, m_1, \ldots, m_{k-1}) \circ \begin{bmatrix} g_{00} & g_{01} & \cdots & g_{0,n-1} \\ g_{10} & g_{11} & \cdots & g_{1,n-1} \\ \vdots & \vdots & & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} = \tag{3.3}$$
$$= m_0 g_0 + m_1 g_1 + \cdots + m_{k-1} g_{k-1}$$

3.1.2. Códigos bloque en forma sistemática

Una propiedad deseable de los códigos bloques lineales es la estructura sistemática, esta consiste en la representación de los (n-k) bits de redundancia o paridad seguidos de la palabra de información de k bits. En la siguiente figura se muestra la estructura sistemática comentada.

Figura 3.1: *Distribución de los bits en una palabra código de forma sistemática*

Un código bloque lineal y sistemático se representa de manera única mediante una matriz generadora de la siguiente forma:

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{bmatrix} = \underbrace{\begin{pmatrix} p_{00} & p_{01} & \cdots & p_{p,n-k-1} \\ p_{10} & p_{11} & \cdots & p_{1,n-k-1} \\ \vdots & \vdots & & \vdots \\ p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} \end{pmatrix}}_{\text{submatriz } \boldsymbol{P} \text{ k x (n k)}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{pmatrix}}_{\text{submatriz } \boldsymbol{I} \text{ k x k}} \tag{3.4}$$

Esta, cual de manera compacta, se corresponde con:

$$G = \begin{bmatrix} P & I_k \end{bmatrix} \tag{3.5}$$

Así, los componentes de la palabra código se pueden desglosar de la siguiente forma:

$$\mathrm{c_{n-k+i}} = \mathrm{m_i} \tag{3.6}$$

$$\mathrm{c_j} = \mathrm{m_0 p_0 j} + \mathrm{m_1 p_1 j} + \cdots + \mathrm{mk} - \mathrm{1 p_{k-1,j}} \qquad 0 <= \mathrm{j} < \mathrm{n} - \mathrm{k} \tag{3.7}$$

Estas n-k ecuaciones se denominan *parity-check equations* del código.

3.1.3. Parity Check Matrix H

Matriz asociada a todo código bloque lineal, esto es, para cada matriz generadora G de dimensiones **k x n** existe una matriz **H** de dimensiones (**n-k**) x **n** con **n- k** filas linealmente independientes de tal manera que, cualquier vector generado por el espacio de las filas de **G** es ortogonal con las filas de **H** y viceversa. De este modo, una n-upla c es palabra código dentro del código C(**n,k**) generado por la matriz **G** si, y solo si $\mathbf{c} \cdot \mathbf{H}^T = 0$. El código es el subespacio nulo de **H**; es más, las 2^{k-n} combinaciones lineales posibles de las filas de la matriz **H** forman un código lineal C_d(**n, n-k**). Este código es el espacio nulo de C(**n-k**) generado por la matriz **G**. Esto es, para cada **c** perteneciente a C y cualquier **w** perteneciente a $C_d, \mathbf{c} \cdot \mathbf{w} = 0$.

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}}_{\text{submatriz } \boldsymbol{I} \text{ (n-k) x (n-k)}} \underbrace{\begin{bmatrix} p_{00} & p_{01} & \cdots & p_{p,n-k-1} \\ p_{10} & p_{11} & \cdots & p_{1,n-k-1} \\ \vdots & \vdots & & \vdots \\ p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-k-1} \end{bmatrix}}_{\text{submatriz } \boldsymbol{P}^T \text{ (n-k) x k}} = \begin{bmatrix} \boldsymbol{I}_{n-k} & \boldsymbol{P}^T \end{bmatrix} \tag{3.8}$$

Donde $\mathbf{P}^T$es la traspuesta de la matriz de paridad **P**. Sea $\mathbf{h}_j$ la fila j-ava de **H** y sea $\mathbf{g}_i$ la fila i-ava de la matriz **G**, tenemos:

$$\mathbf{g}_i \cdot \mathbf{h}_j = p_i j + p_i j = 0; \qquad 0 <= i < k, 0 <= j < n - k \tag{3.9}$$

De lo cual se desprende:

$$\mathbf{G} \cdot \mathbf{H}^T = \mathbf{0} \tag{3.10}$$

y dado que $\mathbf{c} \cdot \mathbf{H}^T = \mathbf{0}$, se obtiene:

$$c_j + p_0 j m_0 + p_1 j m_1 + \cdots + p_{k-1,j} m_{k-1} = 0 \tag{3.11}$$

Lo que equivale a:

$$c_j = p_0 j m_0 + p_1 j m_1 + \cdots + p_{k-1,j} m_{k-1}; \qquad 0 <= j < n-k \tag{3.12}$$

Así, un código lineal (**n,k**) queda completamente definido a partir de su matriz **H**.

3.2. Síndrome y detección de errores

Considerando $c = (c_0, c_1, \ldots, c_{n-1})$ como una secuencia de mensaje codificada sin errores y $r = (r_0, r_1, \ldots, r_{n-1})$, la secuencia recibida a través del canal, podemos definir el vector de error como:

$$\mathbf{e} = \mathbf{r} + \mathbf{c} = (e_0, e_1, \ldots, e_{n-1}) \tag{3.13}$$

Donde $\boldsymbol{e_i} = 1$ *para* $\boldsymbol{r_i} \neq \boldsymbol{c_i}, y\, \boldsymbol{e_i} = 0$ *para* $\boldsymbol{r_i} = \boldsymbol{e_i}$.

Así, donde haya un 1 significará que en esa posición existe un error generado por el ruido de canal. Aplicando las propiedades de un cuerpo GF(2) tenemos:

$$\mathbf{r} = \mathbf{e} + \mathbf{c} \tag{3.14}$$

En recepción, lógicamente, no se conoce ni $\boldsymbol{e}$ ni $\boldsymbol{c}$. Por ello, una vez se recibe la secuencia $\boldsymbol{r}$, se ha de determinar si se han producido errores, intentar localizarlos y corregirlos.

Un mecanismo de detección de errores basado en la matriz H se implementa de la siguiente forma;

$$\mathbf{s} = \mathbf{r} \cdot \mathbf{H}^T = (s_0, s_1, \ldots, s_{n-k-1}) \tag{3.15}$$

Esta (**n-k**)-upla se denomina *síndrome* de $\boldsymbol{r}$. Este vector tiene la propiedad de ser 0 si, y sólo si $\boldsymbol{r}$ es una palabra código, y $s \neq 0$ si, y sólo si $\boldsymbol{r}$ no lo es. Es posible que ciertos patrones de errores no sean detectables; esto es, que se hayan producido errores y $\boldsymbol{s}$=0. Esto ocurre cuando el vector de error $\boldsymbol{e}$ es idéntico a una palabra código distinta de 0. Por ello, habrá exactamente 2^{k-1} errores no detectables.

Si seguimos la definición de síndrome

$$\mathbf{s} = \mathbf{r} \cdot \mathbf{H}^T = (\mathbf{c} + \mathbf{e})\mathbf{H}^T = \mathbf{c} \cdot \mathbf{H}^T + \mathbf{e} \cdot \mathbf{H}^T = \mathbf{e} \cdot \mathbf{H}^T \tag{3.16}$$

De modo que, multiplicando $\boldsymbol{e}$ con la matriz de paridad tenemos la siguiente relación lineal entre el síndrome y los dígitos de error

$$\begin{aligned}
s_0 &= e_0 + e_{n-k}p_0 0 + e_{n-k+1}p_1 0 + \cdots + e_{n-1}p_{k-1,0} \\
s_0 &= e_0 + e_{n-k}p_0 0 + e_{n-k+1}p_1 0 + \cdots + e_{n-1}p_{k-1,0} \\
&\vdots \\
s_{n-k-1} &= e_{n-k-1} + e_{n-k}p_{0,n-k} + e_{n-k+1}p_{1,n-k-1} + \cdots + e_{n-1}p_{k-1,n-k-1}
\end{aligned} \tag{3.17}$$

Desafortunadamente, existen 2^k soluciones para estas n-k ecuaciones lineales. Por ello, existen 2k patrones de error que generan el mismo síndrome; el verdadero vector de error $\boldsymbol{e}$ es tan solo uno de ellos. Si nos encontramos con un canal BSC, el vector de error más probable es aquél que tenga el menor número

3.3. Distancia mínima de un código bloque

Este parámetro determina las capacidades de detección y corrección de errores del código bloque. Una definición importante relacionada con la distancia de un código es el *peso* o *peso de hamming* $\boldsymbol{w}(\boldsymbol{c})$, que se define como el número de componentes de $\boldsymbol{c}_i$ distintos de cero $\boldsymbol{c}_i \neq 0$ de un vector $\boldsymbol{c}$ de tamaño 1 x n, de modo que el vector $\boldsymbol{c}$ =(1 0 0 1 0 1) tiene un peso de hamming $\boldsymbol{w}(\boldsymbol{c})$=4. Otra medida de distancia importante es la distancia de hamming entre dos vectores; esto es, la distancia de hamming entre dos vectores $c_1 = (c_{01}, c_{11}, \ldots, c_{n-1,1})$ y $c_2 = (c_{02}, c_{12}, \ldots, c_{n-1,2})$, $d(c_1, c_2)$ es el número de posiciones en las cuales sus componentes difieren. Así, la distancia de hamming entre $\boldsymbol{v}$ =(1 0 0 1 0 1 0) y $\boldsymbol{w}$ =(0 0 0 1 1 1 0) es 3.

De acuerdo con estas definiciones, se puede verificar que:

$$d(\mathbf{c}_i, \mathbf{c}_j) = \mathbf{w}(c_i + c_j) \tag{3.18}$$

La mínima distancia hamming de un código bloque se obtiene de la distancia mínima evaluada a través de todas las palabras del código:

$$d_{min} = min\left\{d(\mathbf{c}_i, \mathbf{c}_j); c_i, c_j \in C; c_i \neq c_j\right\} \tag{3.19}$$

De acuerdo con los visto y las propiedades de linealidad se deduce:

$$d_{min} = min\left\{d(\mathbf{c}_i, \mathbf{c}_j); c_i, c_j \in C; c_i \neq c_j\right\} = min\left\{w(c_m); c_m \in C; c_m \neq 0\right\} \tag{3.20}$$

Teorema 1 *La mínima distancia de un código bloque es igual al mínimo peso de sus palabras código distintas de la palabra código y viceversa.*

Existe también una relación entre la mínima distancia de un código y su matriz de chequeo de paridad H.

Teorema 2 *Sea C un código lineal (n,k) con matriz de chequeo de paridad* $\boldsymbol{H}$ *. Por cada palabra código con peso de hamming l, existen l columnas de* $\boldsymbol{H}$ *tales que el vector suma de las l columnas es igual al vector* $\boldsymbol{0}$ *. Dicho de otro modo, si existen l columnas de* $\boldsymbol{H}$ *cuya suma genera el vector* $\boldsymbol{0}$*, entonces existe una palabra código con peso de hamming l dentro de C.*

De este teorema se desprende que el mínimo peso del código C es igual al mínimo número de columnas de H que generan el vector $\mathbf{0}$.

3.4. Capacidad de detección y corrección de un código bloque

Como se ha dicho en el apartado anterior, la distancia mínima de un codificador en bloque es el menor número de elementos del que difieren entre sí dos palabras código de ese mismo codificador. De modo que hay al menos dos palabras código válidas las cuales tienen d_{min} componentes distintos entre sí. Hemos visto también que una secuencia recibida a través de un canal ruidoso $\boldsymbol{r}$ se puede desglosar en la secuencia codificada c más un vector de error $\boldsymbol{e}$. De

este modo la distancia entre la secuencia recibida d($\boldsymbol{r},\boldsymbol{v}$)= l es el número de errores que se han producido por el efecto del canal. Así, si $l < d_{min}$ estamos seguro que se ha producido un error, porque no hay ninguna palabra código que difiera de otra distinta menos que d_{min}. Si $l = d_{min}$, no podemos asegurar que haya un error, pues hay al menos dos palabras código que difieren en ese número de elementos. Por ello, se dice que la capacidad de detección de un codificador en bloque de distancia mínima d_{min} es $d_{min} - 1$. De este modo, aseguramos que la secuencia recibida contiene errores, pero el límite $d_{min} - 1$ es el peor de los casos, pues no todas las secuencias con distancia d_{min} serán palabras código válidas y, en consecuencia no su síndrome será distinto de 0. De hecho, un codificador lineal (n,k) contiene $2^n - 1$ posibles secuencias de las cuales tan solo $2^k - 1$ son secuencias código válidas distintas a 0. Por ello, el codificador será capaz de detectar $2^n - 2^k$ vectores de error de longitud n. Se define, pues, la capacidad de que se produzca un error no detectado como:

$$P_u = \sum_{i=1}^{n} A_i p^i (1-p)^{n-i} \tag{3.21}$$

donde A_i es el número de palabras código con peso i dentro de C y p es la función de probabilidad del canal BSC. Lógicamente, los valores de A_1 hasta A_{dmin-1} son 0.

Ejemplo 1 *Tengamos un código bloque lineal (7,4) con* $A_0 = 1, A_1 = A_2 = 0, A_3 = 7, A_4 = 7, A_5 = A_6 = 0, A_7 = 1$. *La probabilidad de errores no detectados es:*

$$P_u(E) = 7p^3(1-p)^4 + 7p^4(1-p)^3 + p^7 \tag{3.22}$$

Para determinar la capacidad correctora de un código bloque C (n,k) de distancia mínima d_{min}, definiremos el entero t:

$$2t + 1 \leq d_{min} \leq 2t + 2 \tag{3.23}$$

Sean $\boldsymbol{c}$ y $\boldsymbol{r}$ las secuencias transmitidas y recibidas respectivamente, y sea $\boldsymbol{w}$ otra palabra dentro de C. Las distancias de hamming de $\boldsymbol{c}$, $\boldsymbol{r}$ y $\boldsymbol{w}$ cumplen

la siguiente inecuación:

$$d(\boldsymbol{c}, \boldsymbol{r}) + d(\boldsymbol{w}, \boldsymbol{r}) \geq d(\boldsymbol{c}, \boldsymbol{w}) \tag{3.24}$$

Si ahora nos ponemos en el caso de tener t' errores durante la transmisión de $\boldsymbol{c}$, $\boldsymbol{r}$ y $\boldsymbol{c}$ difieren en t' posiciones d($\boldsymbol{c}$,$\boldsymbol{r}$)=t'. Como $\boldsymbol{c}$ y $\boldsymbol{w}$ son palabras código dentro de C:

$$d(\mathbf{c}, \mathbf{w}) \geq d_{min} \geq 2t + 1 \tag{3.25}$$

Combinando las dos últimas ecuaciones obtenemos la siguiente inecuación

$$d(\mathbf{w}, \mathbf{r}) \geq 2t + 1 - t' \tag{3.26}$$

si $t' \leq t$ entonces:

$$d(\mathbf{w}, \mathbf{r}) > t \tag{3.27}$$

La desigualdad de arriba nos viene a decir que si se producen menos de t errores, la secuencia recibida $\boldsymbol{r}$ está más próxima a la secuencia transmitida $\boldsymbol{c}$ que a cualquier otra palabra código $\boldsymbol{w}$. Esto significa, también, que la probabilidad P($\boldsymbol{r} \mid \boldsymbol{c}$) es mayor que la probabilidad P($\boldsymbol{r} \mid \boldsymbol{w}$) para $\mathbf{w}$ distinto de $\mathbf{c}$. De este modo, se implementa el criterio de máxima verosimilitud donde $\mathbf{r}$ es decodificado en $\mathbf{c}$. Se puede demostrar que el código bloque puede corregir satisfactoriamente t errores siendo t:

$$\left\lfloor \frac{d_{min}-1}{2} \right\rfloor \tag{3.28}$$

El parámetro t es pues la capacidad correctora de errores del código. Al igual que pasaba con la detección de errores, para la corrección de éstos, t es también la cota por la cual se asegura que se corregirán todos los errores. Pero en realidad pasa que se pueden corregir más errores con cierto grado de fiabilidad. La probabilidad que se produzca una corrección errónea viene dada por:

$$P(E) \geq \sum_{i=t+1}^{n} \binom{n}{i} p^i (1-p)^{n-i} \tag{3.29}$$

3.5. Códigos Hamming

Durante los años 40, Richard W. Hamming, que trabajaba en los laboratorios Bell empezó a crear un seguido de algoritmos para poder evitar los múltiples errores de lectura que cometía la computadora del modelo V de

Bell, una computadora basada en relés con velocidad de proceso de herz. Así, en 1950 Hamming publicó la primera clase de códigos bloque para la corrección de errores, el llamado código Hamming.

Longitud del código	$n = 2^m - 1$
Número de bits de información	$k = 2^m - m - 1$
Número de bits de chequeo de paridad	n-k=m
Capacidad correctora de error	$t = 1(d_{min} = 3)$

La matriz de chequeo de paridad H de este tipo de códigos está formada por columnas distintas al vector 0 de m bits. En su representación sistemática la matriz H tiene la siguiente forma:

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{I_m} & \boldsymbol{Q} \end{bmatrix} \tag{3.30}$$

Donde $\boldsymbol{I}_m$ es la matriz identidad de tamaño $m\ x\ m$ y la submatriz Q consiste en $2^m - m - 1$ columnas que son m-uplas de peso mayor o igual a 2. En el caso más simple de m = 3 tenemos:

$n = 2^3 - 1 = 7$
$k = 2^3 - 3 - 1 = 4$
$n - k = m = 3$
$t = 1(d_{min} = 3)$

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

La matriz generadora puede ser construida en su forma sistemática de la siguiente manera:

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{Q^T} & \boldsymbol{I}_{2^m-1-1} \end{bmatrix} \tag{3.31}$$

Si se observa la matriz H, fácilmente podemos encontrar tres columnas que sumadas nos lleven al vector nulo; en cambio, con dos no es posible. De ahí que la distancia mínima del código sea 3. Los códigos hamming tienen la propiedad de corregir un error y no más de uno. Como curiosidad, decir que los códigos capaces de corregir exactamente t errores se denominan códigos perfectos. Son códigos muy inusuales; de hecho, tan sólo hay otro código

no trivial que sea perfecto, el código Golay (23,12). Como código perfecto, la matriz estándar del código Hamming tiene como cosets líderes todos los patrones de error t.

Así, el método para decodificar estos códigos es mediante su síndrome de la siguiente manera:

1. Si el síndrome $\mathbf{s}$ es zero, damos por hecho que no se ha producido error alguno.

2. Si $\mathbf{s}$ tiene un número impar de unos damos por hecho que se ha producido un único error. Buscamos el patrón de error que se corresponde con $\mathbf{s}$ y se lo añadimos al vector recibido.

3. Si $\mathbf{s}$ tiene un número par de unos asumimos que se ha producido un número de errores que supera la capacidad correctora

3.6. Códigos Cíclicos

Estos códigos representan una importante subclase de los códigos bloque lineales. Este tipo de códigos se pueden implementar fácilmente mediante registros de desplazamiento y, además, tienen una estructura algebraica que facilita el proceso de decodificación.

3.6.1. Descripción

Dado un vector código de n componentes $\boldsymbol{c} = (c_0, c_1, ..., c_{n-1}))$, una rotación hacia la derecha genera otro vector código. Si rotamos i veces el vector código original obtenemos:

$$\mathbf{c}^{(i)} = (c_{n-i}, c_{n-i+1}, \ldots, c_{n-1}, c_0, c_1, \ldots, c_{n-i-1}) \tag{3.32}$$

Donde se puede observar que desplazando $\boldsymbol{c}$ i veces hacia la derecha es equivalente a desplazar $\boldsymbol{c}$ n-i veces hacia la izquierda.

Definición 1 *Un código lineal C (n,k) es un código cíclico si cada rotación cíclica de una palabra código de C es también una palabra código de C*

3.6.2. Representación polinómica de los códigos cíclicos.

Con el fin de desarrollar las propiedades algebraicas de un código cíclico trataremos los componentes de las palabras código $\boldsymbol{c} = (c_0, c_1, ..., c_{n-1}))$ como coeficientes de un polinomio:

$$\mathbf{c}(X) = c_0 + c_1 X + c_2 X^2 + \cdots + c_{n-1} X^{n-1} \tag{3.33}$$

La correspondencia entre la palabra código c y el polinomio c(X) es uno a uno. Dados los polinomios $c_1(X)$ y $c_2(X)$ ambas representaciones polinómicas de las palabras código c_1 y c_2, se definen las operaciones de suma y multiplicación de éstas como:

$$c_1(X) + c_2(X) = c_0 1 + c_0 2 + (c_1 1 + c_1 2)X + \cdots + (c_{n-1,1} + c_{n-1,2})X^{n-1} \tag{3.34}$$

$$c_1(X) \cdot c_2(X) = c_0 1 \cdot c_0 2 + (c_0 1 \cdot c_1 2 + c_0 2 \cdot c_1 1)X + \cdots + (c_{n-1,1} \cdot c_{n-1,2})X^{2(n-1)} \tag{3.35}$$

Las operaciones de multiplicación y adición de las expresiones anteriores están definidas bajo un campo finito módulo 2, y ambas cumplen las propiedades distributiva, asociativa y conmutativa. Para definir la división, tomaremos dos palabras código c_1 y c_2 de las cuales $c_2(X) \neq 0$. La operación entre los polinomios c_1 y c_2 se define a través de la existencia de dos únicos polinomios dentro del mismo campo q(X) y r(X), cociente y residuo respectivamente, con la estructura siguiente:

$$c_1(X) = q(X)c_2(X) + r(X) \tag{3.36}$$

3.6.3. Polinomio Generador

Dado el código polinómico correspondiente con la palabra código $\boldsymbol{c}^{(i)}$

$$\boldsymbol{c}^{(i)} = c_{n-i} + c_{n-i+1}X + \cdots + c_{n-1}X^{i-1} + c_0X^i + c_1X^{i+1} + \cdots + c_{n-i-1}X^{n-1} \tag{3.37}$$

manipulando la ecuación anterior llegamos a la expresión siguiente:

$$X^i\mathbf{c}(X) = \mathbf{q}(X)(X^n + 1) + \mathbf{c}^{(i)}(X) \tag{3.38}$$

donde

$$\mathbf{q}(X) = c_{n-i} + c_{n-i+1}X + \cdots + c_{n-1}X^{i-1} \tag{3.39}$$

Así, se pude observar que $c^{(i)}(X)$ es el residuo de dividir $X^i\boldsymbol{c}(x)$ entre $X^i + 1$.

A continuación demostraremos que el polinomio de grado mínimo dentro de C es único. Además el valor constante de este polinomio c_0 debe ser 1. Sea $g(X) = g_0 + g_1X + \cdots + g_{r-1}X^{r-1} + X^r$, un polinomio de grado mínimo en C. Sea $g'(X) = g'_0 + g'_1X + \cdots + g'_{r-1}X^{r-1} + X^r$, si sumamos estos dos polinomios obtenemos $g'(X) + g(X) = (g_0 + g'_0) + (g_1 + g'_1)X + \cdots + (g'_{r-1} + g'_{r-1})X^{r-1}$ que, por propiedades de linealidad, también pertenece a C y es de grado menor a r, así el polinomio de grado mínimo es único. Si ahora suponemos que $g_0 = 0$, tenemos; $g(X) = g_1X + \cdots + g_{r-1}X^{r-1} + X^r$. Desplazando el código una posición hacia la izquierda obtenemos $g_1 + \cdots + g_{r-1}X^{r-2} + X^{r-1}$, que vuelve a ser de grado inferior a r.

Así, el polinomio de grado mínimo de un código cíclico C tiene la siguiente estructura:

$$\mathbf{g}(X) = 1 + g_1X + g_2X^2 + \cdots + g_{r-1}X^{r-1} + X^r \tag{3.40}$$

Se puede demostrar que cualquier código polinómico C(X) en un código cíclico (n,k) puede ser expresado de la siguiente forma:

$$C(X) = m(X)g(X) = (m_0 + m_1X + \cdots + m_{k-1}X^{k-1})g(X) \tag{3.41}$$

Donde $m_0m_1 \ldots m_{k-1}$ son bits del mensaje a codificar. Hay dos propiedades importantes que cabe resaltar, aunque no las demostraremos en este repaso a los códigos cíclicos. La primera de ellas cumple que el polinomio generador de un código cíclico $(\boldsymbol{n,k})$ es un factor de $X^n + 1$. La segunda afirma que cualquier polinomio que sea factor de $X^n + 1$ genera un codigo cíclico $(\boldsymbol{n,k})$. De aquí se puede demostrar que X$\mathbf{g}$(X) es también un código polinómico en C. Dado que un polinomio generado g(X) es también posible mostrar el código en forma sistemática, los k dígitos de la derecha será el mensaje inalterado, y los n- k dígitos de la izquierda, los bits de paridad. Supongamos el mensaje a ser codificado $m = (m_0, m_1, \ldots, m_{k-1})$, el cual, puesto de forma polinómica nos queda como $m(X) = m_0 + m_1X + \cdots + m_{k-1}X^{k-1}$. Si multiplicamos $m(X)$ por X^{n-k} y lo dividimos por el polinomio generador nos queda la siguiente expresión:

$$X^{n-k}\mathbf{m}(X) = \mathbf{a}(X)\mathbf{g}(X) + \mathbf{b}(X) \tag{3.42}$$

Al ser el grado de $g(X)$ $n-k$, el grado del residuo $b(X)$ será como mucho $n-k-1$, y reordenando los polinomios obtenemos:

$$\mathbf{b}(X) + X^{n-k}\mathbf{m}(X) = \mathbf{a}(X)\mathbf{g}(X) \tag{3.43}$$

Por la forma de la expresión vemos que $b(X) + X^{n-k}m(X)$ es una palabra código, la cual tiene la siguiente estructura

$$\begin{aligned}&\mathbf{b}(X) + X^{n-k}\mathbf{m}(X) = b_0 + b_1X + \ldots \\ &+ b_{n-k-1}X^{n-k-1} + m_0X^{n-k} + m_1X^{n-k+1} + \cdots + m_{k-1}X^{n-1}\end{aligned} \tag{3.44}$$

Esto es la palabra código en forma sistemática:

$$(b_0, b_1, \ldots, n_{n-k-1}, m_0, m_1, \ldots, m_{k-1}) \tag{3.45}$$

los primeros $b_0, b_1, \ldots, b_{n-k-1}$ son los **n-k** bits de chequeo de paridad, los cuales se corresponden con el residuo de dividir $X^{n-k}m(X)$ entre el polinomio generador $\mathbf{g}(X)$.

3.6.4. Matriz generadora de un código cíclico.

Como se ha indicado anteriormente, un código cíclico $C(n,k)$ generado por un polinomio generador $g(X)$ se extiende en k códigos polinómicos $\mathbf{g}(X), X\mathbf{g}(X), \ldots X^{n-k}\mathbf{g}(X)$. Colocando estos códigos polinómicos como filas de una matriz obtenemos la matriz generadora del código polinómico.

$$\mathbf{G} = \begin{bmatrix} g_0 & g_1 & g_2 & \ldots & g_{n-k} & 0 & 0 & \ldots & 0 \\ 0 & g_0 & g_1 & \ldots & g_{n-k-1} & g_{n-k} & 0 & \ldots & 0 \\ \vdots & \vdots & & & \vdots & \vdots & & & \vdots \\ 0 & 0 & & \ldots & g_0 & g_1 & g_2 & \ldots & g_{n-k} \end{bmatrix} \tag{3.46}$$

Donde $g_0 = g_{n-k} = 1$.

De manera sistemática, la matriz generadora se obtendría dividiendo X^{n-k+i} entre $\mathbf{g}(X)$ para $i = 0, 1, \ldots, k-1$:

$$X^{n-k+i} = \mathbf{a}_i(X)\mathbf{g}(X) + \mathbf{b}_i(X) \tag{3.47}$$

donde $\mathbf{b}_i(X) + X^{n-k+i}$ son polinomios código que, ordenados en una matriz de dimensión **k x n**, dan lugar a la matriz G de forma sistemática:

$$\mathbf{G} = \begin{bmatrix} b_{00} & b_{01} & b_{02} & \ldots & b_{0,n-k-1} & 1 & 0 & 0 & \ldots & 0 \\ b_{10} & b_{11} & b_{12} & \ldots & b_{1,n-k-1} & 0 & 1 & 0 & \ldots & 0 \\ b_{20} & b_{21} & b_{22} & \ldots & b_{2,n-k-1} & 0 & 0 & 1 & \ldots & 0 \\ & \vdots & & & \vdots & & & & & \vdots \\ b_{k-1,0} & b_{k-1,1} & b_{k-1,2} & \ldots & b_{k-1,n-k-1} & 0 & 0 & 0 & \ldots & 1 \end{bmatrix} \tag{3.48}$$

La matriz de chequeo de paridad de C tiene la siguiente forma:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & b_{00} & b_{01} & b_{02} & \dots & b_{0,n-k-1} \\ 0 & 1 & 0 & \dots & 0 & b_{10} & b_{11} & b_{12} & \dots & b_{1,n-k-1} \\ 0 & 0 & 1 & \dots & 0 & b_{20} & b_{21} & b_{22} & \dots & b_{2,n-k-1} \\ & \vdots & & & \vdots & & & & & \vdots \\ 0 & 0 & 0 & \dots & 1 & b_{k-1,0} & b_{k-1,1} & b_{k-1,2} & \dots & b_{k-1,n-k-1} \end{bmatrix} \tag{3.49}$$

3.6.5. Codificación del síndrome y detección de errores.

Como ya se ha visto en secciones anteriores, el primer paso para obtener el síndrome en un código lineal es multiplicar el vector recibido por la matriz de paridad, $\mathbf{s} = \mathbf{r}\mathbf{H}^T$. Si el código recibido se corresponde con el transmitido, entonces el síndrome es cero. Para el caso de un código cíclico en forma sistemática, el síndrome se puede obtener de forma sencilla como residuo de la división entre $\mathbf{g}(X)$:

$$\mathbf{r}(X) = \mathbf{a}(X)\mathbf{g}(X) + \mathbf{s}(X) \tag{3.50}$$

el residuo $\mathbf{s}(X)$ es un polinomio de grado igual o menor que ***n- k-1***. Los $n-k$ coeficientes de $\mathbf{s}(X)$ se corresponden al síndrome $\mathbf{s}$. La idea es sencilla, si $\mathbf{r}(X)$ es un polinomio código generado a partir de $\mathbf{g}(X)$, su residuo debería ser 0 al ser dividido entre $\mathbf{g}(X)$.

Teorema 3 *Sea s(X) el síndrome del polinomio recibido* $\boldsymbol{r}(X) = \boldsymbol{r}_0 + \boldsymbol{r}_1 X + \cdots + \boldsymbol{r}_{n-1}X^{n-1}$*, entonces, el residuo* $\boldsymbol{s}^{(1)}(X)$*resultado de de dividir* $X\boldsymbol{s}(X)$ *entre el polinomio generador* $g(X)$ *es el síndrome* $\boldsymbol{r}^{(1)}(X)$*, el cual es el desplazamiento cíclico de* $\boldsymbol{r}(X)$

Este teorema es muy útil para generar una tabla de síndromes del código sin la necesidad de recibir todas las secuencias código. Así, para obtener el síndrome $\mathbf{s}^{(i)}(X)$ de $\mathbf{r}^{(i)}(X)$ tan sólo debemos desplazar i-veces el registro inicial $\mathbf{s}(X)$.

3.6.6. Decodificación de los códigos cíclicos

La decodificación de los códigos cíclicos sigue la misma estructura de asociar error con síndrome y síndrome con vector recibido, ya que, como se demostró en secciones anteriores, el síndrome del vector recibido es el mismo que el síndrome del error producido sobre éste. En el caso de los códigos cíclicos, éstos tienen una estructura algebraica que facilita el computo; así, no es necesario la reconstrucción de una tabla, sino que se puede decodificar dígito a dígito mediante un proceso de cálculo de síndrome y desplazamiento.

Una vez recibimos el mensaje a través del canal, comprobamos si su síndrome $\mathbf{s}(X)$ se corresponde con el síndrome de un error en el primer dígito, es decir, el de mayor orden $X^{n-1}(\mathbf{e}_{n-1} = 1)$. Si es así, se suma el error a la secuencia recibida, si no es así, se desplaza el vector recibido $r(1)(X) = r_{n-1} + r_0 X + \cdots + r_{n-2}X^{n-1}$, se calcula el síndrome de este código polinómico y se comprueba si ahora sí se corresponde con el síndrome de un error en la última posición. Si resulta que hemos detectado un error en esta posición (esto es, cuando se correspondan los síndromes), entonces corregimos el error mediante la suma $r_{n-1} \oplus e_{n-1}$. Una vez hecho esto, el polinomio recibido nos queda modificado; $\mathbf{r}_1(X) = r_0 X + \cdots + r_{n-2}X_{n-2} + (r_{n-1} \oplus e_{n-1})X^{n-1}$. Se modifica el error en en-1 del síndrome s(X); esto se realiza añadiendo el síndrome de $\mathbf{e}'(X) = X^{n-1}$ a s(X). El resultado de esta suma es el sindrome de $\mathbf{r}_1(X)$. Una vez obtenido el síndrome $\mathbf{s}_1(X)$ se desplazan éste y el código recibido $\mathbf{r}_1(X)$. De este modo tenemos:

$$\mathbf{r}_1^{(1)}(X) = (r_{n-1} \oplus e_{n-1}) + r_0 X + \cdots + r_{n-2}X^{n-1} \tag{3.51}$$

Por procedimientos algebraicos obtenemos:

$$\mathbf{s}_1^{(1)} = \mathbf{s}^{(1)}(X) + 1 \tag{3.52}$$

A continuación se procede a decodificar r_{n-2} de la misma manera que se decodificó r_{n-1}.

Una vez corregidos todos los errores, se rota cíclicamente el polinomio tantas veces como lo hayamos rotado en el cálculo.

Ejemplo 2 .

$s(X)$	e_{n-1}
100	0000001

Secuencia recibida: $\boldsymbol{r} = 0101011$
Calculamos síndromes y comprobamos si se corresponden con un error en el bit más significativo.

n	*secuencia rotada n*	*síndrome*	
0	0101011	101	
1	1010101	001	
2	1101010	010	
3	0110101	100	*Bit de la ultima posicion erroneo*
4	0011010	000	*Ok*
5	0001101	000	*Ok*
6	1000110	000	*Ok*

$\boldsymbol{c} = 0100011$

Comprobamos que en la rotación 3 el síndrome se corresponde con el del error en el bit más significativo. Lo cambiamos y comprobamos que no hay más errores. Finalmente, desplazamos tres posiciones en sentido contrario la secuencia y obtenemos la secuencia transmitida $\boldsymbol{c}$.

3.7. Códigos BCH binarios

Estos códigos deben su nombre a sus creadores Bose, Chaudhuri y Hocquenghem y son una generalización de los códigos Hamming para una corrección con multiples errores. Fueron descubiertos por primera vez en 1959 por Hocquenghem. Un año más tarde, en 1960, serían Bose y Chaudhuri los que se harían con estos códigos sin tener constancia de los trabajos de Hocquenghem. Los códgios BCH binarios estan definidos bajo un cuerpo GF (2), más también existen para cuerpos no binarios GF(q). Estos últimos representan la subclase más importante de los códigos Reed-Solomon descubiertos por Reed y Solomon, también en 1960 de forma independiente de los trabajos de Bose, Chaudhuri y Hocquenghem. En lo que a decodificación se refiere, los

algoritmos más eficientes para decodificar los códigos BCH son el algoritmo iterativo de Berlekamp y el algoritmo de localización de Chien.

Los códigos BCH forman parte de los códigos cíclicos que, como hemos visto, se generan multiplicando el mensaje m(X) por el polinomio generador del código g(X). De este modo, obtenemos que nuestro código polinómico c(X) tiene al menos tantas raíces como g(X). Un polinomio tiene tantas raíces como su grado, aunque no se puedan encontrar todas dentro del campo GF(2), sí las podemos obtener dentro de$GF(2^m)$.

Así, tal como hemos visto en codificaciones anteriores, los códigos BCH también basan su funcionamiento a partir de modelos de detección de síndrome. Al ser las raíces de los códigos las mismas que las de sus polinomios generadores, podemos poner el foco de nuestro diseño en polinomios generadores con raíces conocidas y con propiedades algebraicas que optimicen su decodificación y eficiencia.

Hemos visto también en secciones anteriores que los códigos bloque tienen una capacidad correctora que va acorde con la distancia mínima del código:

$$t = \left\lfloor \frac{d_{min}-1}{2} \right\rfloor \tag{3.53}$$

Así, debemos generar polinomios generadores que nos permitan corregir el número de errores que precisemos. Naturalmente, el tamaño del código crecerá a mayor capacidad correctora.

Vayamos ahora a precisar un poco más la estructura de los códigos BCH.

3.7.1. Códigos cíclicos binarios BCH

Los códigos BCH representan una generalización de los códigos de Hamming, los cuales son capaces de corregir 1 error. El paso hacia la corrección de t errores cualesquiera requiere un proceso de diseño que se basa en obtener polinomios generadores a partir del mínimo común múltiple MCM de los polinomios adecuados.

Para cualquier entero positivo $m \geq 3$ y $t < 2^{m-1}$, existe un código binario BCH con los siguientes parámetros:

Tamaño del bloque:	$n = 2^m - 1$
número de bits de paridad	$n - k \leq mt$
Distancia mínima:	$d_{min} \geq 2t + 1$
Capacidad correctora:	t errores por bloque

El polinomio generador de este código se especifica en términos de sus raíces bajo el campo $GF(2^m)$. Dado α un elemento primitivo dentro de $GF(2^m)$, el polinomio generador que corrige t errores de un código BCH de tamaño $2^m - 1$ es el polinomio de menor grado bajo GF(2) que tenga $\alpha, \alpha^2, \ldots, \alpha^{2t}$.

$$g(\alpha^i) = 0, i = 1, 2, \ldots, 2t \tag{3.54}$$

Si $\Phi_i(X)$ es el polinomio de mínimo de α^i, entonces $g(X)$ es el MCM de $\Phi_1(X), \Phi_2(X)..., \Phi_{2t}(X)$:

$$g(X) = \mathbf{MCM}\left\{\Phi_1(X), \Phi_2(X), \ldots, \Phi_{2t}(X)\right\} \tag{3.55}$$

Como los conjugados de α^i también son raíces de g(X), el polinomio generador se puede formar con índices impares de los polinomios mínimos:

$$g(X) = \mathbf{MCM}\left\{\Phi_1(X), \Phi_3(X), \ldots, \Phi_{2t-1}(X)\right\} \tag{3.56}$$

Un código BCH de $t = 1$ es un código Hamming.

Ejemplo 3 *Teniendo en cuenta que* $p(X) = 1 + X + X^4$ *es un polinomio primitivo, la siguiente tabla nos muestra una lista de polinomios mínimos de los elementos en GF(4) generados por* $p(X)$*:*

Raíces conjugadas	*Polinomios mínimos*
0	X
1	$X + 1$
$\alpha, \alpha^2, \alpha^8$	$X^4 + X + 1$
$\alpha^3, \alpha^6, \alpha^9, \alpha^{12}$	$X^4 + X^3 + X^2 + X + 1$
α^5, α^{10}	$X^2 + X + 1$
$\alpha^7, \alpha^{11}, \alpha^{13}, \alpha^{14},$	$X^4 + X^3 + 1$

Observamos que los polinomios mínimos para $\alpha, \alpha^3 y \alpha^5$ *son:*

$$\Phi_1(X) = 1 + X + X^4$$
$$\Phi_3(X) = 1 + X + X^2 + X^3 + X^4$$
$$\Phi_5(X) = 1 + X + X^2$$

Así, un código BCH que corrija 2 errores en un bloque de tamaño $n = 2^4 - 1 = 15$*, tendrá como polinomio generador a:*

$$g(X) = MCM\left\{\Phi_1(X), \Phi_3(X)\right\} \tag{3.57}$$

Al ser ambos polinomios irreducibles y distintos, tenemos:

$$g(X) = \Phi_1(X)\Phi_1(X) = (1+X+X^4)(1+X+X^2+X^3+X^4) = 1+X^4+X^6+X^7+X^8$$

Este es el polinomio generador del código BCH $C_{BCH}(15, 7)$*, con distancia de Hamming* $d_{min} \geq 5$*. Como se puede observar, el número de elementos que tiene el polinomio generador es 5, así que su peso mínimo será 5,* $d_{min} = 5$*.*

Si lo que queremos es un código que corrija tres errores, necesitaremos una distancia mínima mayor. Construimos, de este modo el polinomio generador añadiéndole el siguiente polinomio mínimo de la tabla $\Phi_5(X)$:

$$g(X) = MCM\left\{\Phi_1(X), \Phi_3(X), \Phi_5(X)\right\} = 1 + X + X^2 + X^4 + X^5 + X^8 + X^{10}$$

Este polinomio generador crea un código BCH $C_{BCH}(15, 5)$ capaz de corregir $t = 3$ errores, con distancia mínima $d_{min} \geq 7$. Al ser de nuevo 7 el peso del polinomio generador, la distancia mínima es exactamente 7.

Como resultado de la definición de un código BCH de longitud $n = 2^m -$ 1capaz de corregir t errores, tanto $\alpha, \alpha^2, \ldots, \alpha^{2t}$, como sus conjugados son

raíces. Cualquier código polinómico $C(X) = c_0 + c_1 + \cdots + c_{n-1}$ de $C_{BCH}(n,k)$ tiene el elemento primitivo α^i como raíz:

$$c(\alpha^i) = c_0 + c_1\alpha^i + \cdots + c_{n-1}\alpha^{i(n-1)} = 0 \tag{3.58}$$

De forma matricial:

$$\begin{pmatrix} c_0, & c_1, & \ldots, & c_{n-1} \end{pmatrix} \cdot \begin{bmatrix} 1 \\ \alpha^i \\ \alpha^{2i} \\ \vdots \\ \alpha^{(n-1)i} \end{bmatrix} = 0 \tag{3.59}$$

Así, se puede formar la siguiente matriz:

$$\mathbf{H} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \ldots & \alpha^{n-1} \\ 1 & \alpha^2 & (\alpha^2)^2 & (\alpha^2)^3 & \ldots & (\alpha^2)^{n-1} \\ 1 & \alpha^3 & (\alpha^3)^2 & (\alpha^3)^3 & \ldots & (\alpha^3)^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \alpha^{2t} & (\alpha^{2t})^2 & (\alpha^{2t})^3 & \ldots & (\alpha^{2t})^{n-1} \end{bmatrix} \tag{3.60}$$

Entonces, si c es un vector de código, se cumple:

$$\mathbf{c} \cdot H^T = \mathbf{0} \tag{3.61}$$

Así, H es la matriz de chequeo de paridad del código. Tambiéns se observa que para j e i, tal que α^j es el conjugado de α^i, $c(\alpha^j) = 0$ si, y solo si $c(\alpha^i) = 0$; esto es, si el producto vectorial entre $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ y la fila i de H es 0. Entonces, el producto vectorial de $\mathbf{c}$ con la fila j de H también es 0. Por ello, la fila $j - ava$ de H se puede omitir. Como resultado, la matriz de chequeo de paridad H puede quedar reducida a:

$$\mathbf{H} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \dots & \alpha^{n-1} \\ 1 & \alpha^3 & (\alpha^3)^2 & (\alpha^3)^3 & \dots & (\alpha^3)^{n-1} \\ 1 & \alpha^5 & (\alpha^5)^2 & (\alpha^5)^3 & \dots & (\alpha^5)^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \alpha^{2t-1} & (\alpha^{2t-1})^2 & (\alpha^{2t-1})^3 & \dots & (\alpha^{2t-1})^{n-1} \end{bmatrix} \tag{3.62}$$

Las entradas de H son elementos dentro de $GF(2^m)$. Así, podemos representar cada elemento en $GF(2^m)$ por $m-uplas$ bajo $GF(2)$. Si cambiamos cada entrada de H con su correspondiente $m-upla$ bajo $GF(2)$ ordenados en columnas, obtenemos una matriz de chequeo de paridad binaria del código. Por ejemplo, la matriz de paridad de un código BCH de longitud $n = 2^4 - 1 = 15$, $C_{BCH}(15, 7)$ con capacidad para corregir $t = 2$ errores, y siendo α un elemento primitivo en $GF(2^4)$, es:

$$\mathbf{H} = \begin{bmatrix} 1 & \alpha & \alpha^2 & \alpha^3 & \alpha^5 & \alpha^6 & \alpha^7 & \alpha^8 & \alpha^9 & \alpha^9 & \alpha^{10} & \alpha^{11} & \alpha^{12} & \alpha^{13} & \alpha^{14} \\ 1 & \alpha^3 & \alpha^6 & \alpha^9 & \alpha^{12} & \alpha^0 & \alpha^3 & \alpha^6 & \alpha^9 & \alpha^{12} & \alpha^0 & \alpha^3 & \alpha^6 & \alpha^9 & \alpha^{12} \end{bmatrix} \tag{3.63}$$

Pasando los valores exponenciales a valores vectoriales (ver tabla A.2) la matriz de chequeo de paridad nos queda:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{3.64}$$

3.7.2. Decodificación de los códigos BCH

Supongamos que recibimos la secuencia $r(X)$ en recepción. Si se han producido errores, se puede desglosar dicha secuencia en:

$$r(X) = c(X) + e(X) \tag{3.65}$$

Siendo $e(X)$ la secuencia de error.

Como los códigos BCH son códigos cíclicos, el primer paso para decodificarlos es hallar el síndrome de la secuencia recibida $r(X)$. Para decodificar un código BCH capaz de corregir t errores, el síndrome será una $2t - upla$:

$$\mathbf{S} = (S_1, S_2, \ldots, S_{2t}) = \mathbf{r} \cdot \mathbf{H}^T \tag{3.66}$$

El componente $i - esimo$ del síndrome $\mathbf{S}$ se corresponde a:

$$S_i = \mathbf{r}(\alpha^i) = \mathbf{e}(\alpha^i) = r_0 + r_1\alpha^i + r_2\alpha^{2i} + \cdots + r_{n-1}\alpha^{(n-1)i}, \ \ para \ \ 1 \leq i \leq 2t \tag{3.67}$$

Otra forma de obtener los síndromes es dividir $r(X)$ por el polinomio mínimo $\Phi_i(X)$ de α^i; así, obtenemos:

$$\mathbf{r}(X) = \mathbf{a}_i(X)\Phi_i(X)\mathbf{b}_i(X) \tag{3.68}$$

Al ser $\Phi_i(\alpha^i) = 0$, el síndrome se corresponde con el residuo de la división

$$S_i = \mathbf{r}(\alpha^i) = \mathbf{b}_i(\alpha^i) \tag{3.69}$$

Ahora, si reordenamos la secuencia código en función de la secuencia recibida y la secuencia de error

$$\mathbf{c}(X) = \mathbf{r}(X) + \mathbf{e}(X) \tag{3.70}$$

De modo que si se han producido errores en el canal de transmisión, no todos los elementos de $e(X)$ serán 0. Así, en las posiciones $X^{j1}, X^{j2}, \ldots, X^{jv}$ distintas a 0 serán las posiciones de $c(X)$ erróneas, por lo que para decodificar el código BCH tendremos que localizar estas posiciones y cambiarles el valor:

$$\mathbf{e}(X) = X^{j1} + X^{j2} + \cdots + X^{jv} \tag{3.71}$$

Como $S_i = e(X)$, obtenemos el siguiente sistema de ecuaciones:

$$\begin{aligned} S_1 &= \alpha^{j1} + \alpha^{j2} + \cdots + \alpha^{jv} \\ S_2 &= (\alpha^{j1})^2 + (\alpha^{j2})^2 + \cdots + (\alpha^{jv})^2 \\ S_3 &= (\alpha^{j1})^3 + (\alpha^{j2})^3 + \cdots + (\alpha^{jv})^3 \\ &\vdots \\ S_{2t} &= (\alpha^{j1})^{2t} + (\alpha^{j2})^{2t} + \cdots + (\alpha^{jv})^{2t} \end{aligned} \tag{3.72}$$

Donde $\alpha^{j1}, \alpha^{j2}, \ldots, \alpha^{jv}$ son desconocidos, de modo que cualquier método que resuelva este sistema de ecuaciones representa un algoritmo de decodificación del código BCH. Por comodidad, definiremos los números *localizadores de error* de la siguiente manera:

$$\beta_l = \alpha^{jl} \tag{3.73}$$

De este modo, las ecuaciones anteriores quedan definidas de la siguiente manera:

$$\begin{aligned} S_1 &= \beta_1 + \beta_2 + \cdots + \beta_v \\ S_2 &= \beta_1^2 + \beta_2^2 + \cdots + \beta_v^2 \\ S_3 &= \beta_1^3 + \beta_2^3 + \cdots + \beta_v^3 \\ &\vdots \\ S_{2t} &= \beta_1^{2t} + \beta_2^{2t} + \cdots + \beta_v^{2t} \end{aligned} \tag{3.74}$$

Definimos el polinomio localizador de error del siguiente modo:

$$\sigma(\mathbf{X}) = (1+\beta_1 X)(1+\beta_2 X)\dots(1+\beta_v X) = \sigma_0 + \sigma_1 X + \sigma_2 X^2 + \cdots + \sigma_v X^v \tag{3.75}$$

Y el polinomio evaluador de errores:

$$\mathbf{W}(X) = \sum_{l=1}^{l=v} e_{lj} \prod_{i=1, i\neq l}^{v} (X - \alpha^{-ji}) \tag{3.76}$$

$$e_{jl} = \frac{W(\alpha^{-ji})}{\sigma'(\alpha^{-ji})} \tag{3.77}$$

Donde $\sigma'(X)$ es la derivada de $\sigma(X)$ respecto a X. Así mismo, obtenemos los síndromes de grado $deg\{S(X)\} \leq 2t-1$ como:

$$\mathbf{S}(X) = s_1 + s_2 X + s_3 X^2 + \cdots + s_{2t} X^{2t-1} = \sum_{j=0}^{j=2t-1} s_{j+1} X^j \tag{3.78}$$

Si $S(X) = 0$, entonces no se ha producido error, o el patrón de error introducido no es detectable.

Las raíces de $\sigma(X)$ son β_1^{-1}, β_2^{-1},...,β_v^{-1}. Los coeficientes de $\sigma(X)$, se relacionan con los números localizadores de error de la siguiente forma:

$$\begin{aligned} \sigma_0 &= 1 \\ \sigma_1 &= \beta_1 + \beta_2 + \cdots + \beta_v \\ \sigma_2 &= \beta_1\beta_2 + \beta_2\beta_3 + \cdots + \beta_{v-1}\beta_v \\ &\vdots \\ \sigma_v &= \beta_1\beta_2\dots\beta_v \end{aligned} \tag{3.79}$$

A su vez, los elementos σ^i's están relacionados con los componentes del síndrome S^j's a través de las *identidades de Newton*:

$$
\begin{aligned}
&S_1 + \sigma_1 = 0 \\
&S_2 + \sigma_1 S_1 + 2\sigma_2 = 0 \\
&S_3 + \sigma_1 S_2 + \sigma_2 S_1 + 3\sigma_3 = 0 \\
&\qquad\vdots \\
&S_\upsilon + \sigma_1 S_{\upsilon-1} + \cdots + \sigma_{\upsilon-1} S_1 + \upsilon\sigma_\upsilon = 0 \\
&S_{\upsilon+1} + \sigma_1 S_\upsilon + \cdots + \sigma_{\upsilon-1} S_2 + \sigma_\upsilon S_1 = 0 \\
&\qquad\vdots
\end{aligned}
\tag{3.80}
$$

donde para el caso binario tenemos que $i\sigma^i$ es σ^i para i impar y 0 para i par.

Ahora ya podemos ofrecer tres pasos básicos para la corrección de los códigos BCH binarios. A pesar de que nos podríamos extender en algoritmos para optimizar el procedimiento de decodificación de estos códigos, este no es el objetivo de este proyecto, por lo que no nos extenderemos hasta este punto.

1. Se calcula el síndrome $\mathbf{S} = (S_1, S_2, \ldots, S_{2t})$ a partir de la secuencia recibida $\mathbf{r}(\mathrm{X})$.
2. Se determina el polinomio localizador de error $\sigma(X)$ a partir de los componentes del síndrome $S_1, S_2, \ldots, S_{2t}$.
3. Se hallan los números localizadores de error $\beta_1, \beta_2, \ldots, \beta_\upsilon$ a partir de las raíces de $\sigma(X)$ y se corrigen los errores en $\mathbf{r}(X)$.

3.8. Códigos Reed-Solomon

Los códigos Reed-Solomon son una subfamilia de los códigos BCH no binarios. La característica principal de estos códigos es que los símbolos estan descritos bajo el campo de Galois GF(q), donde ahora q no es una potencia de 2. Estos códigos se denominan *códigos q-arios*. En efecto, en un código BCH no binario el campo finito de Galois es la potencia de un primo, $q = p^m_{prime}$. La distancia mínima de un código RS es igual al nombre de símbolos de

chequeo de paridad más uno. La decodificación de estos códigos presenta dos fases, la localización del error y la determinación del valor del mismo. Uno de los algoritmos de corrección de error consiste en la *búsqueda de Chien*, presentada en el apéndice B, para describir la corrección de errores BCH binarios. Esta vez, sin embargo, el polinomio evaluador no es trivial en el algoritmo, pues los valores de error no son binarios. Se presentará en este apartado el algoritmo iterativo de descodificación Berlekamp.

3.8.1. Introducción: códigos BCH bajo GF(q)

Una buena forma de introducir los códigos BCH no binarios es, simplemente, hacer una extensión de los códigos binarios, pues las propiedades son similares. Para cualesquiera dos enteros positivos t y m, existe un código q-ario de longitud $n = q^m - 1$, de los cuales, como mucho, habrá $2mt$ elementos de paridad y capaz de corregir t errores o menos. Sea α un elemento primitivo en $GF(q^m)$, el polinomio generador capaz de corregir t errores $\mathbf{g}(X)$ de un código BCH $q-ario$ es el polinomio de menor grado bajo $GF(q)$ que tiene como raíces a $\alpha, \alpha^2, \ldots, \alpha^{2t}$. Entonces, si tenemos a $\Phi_i(X)$ como polinomio mínimo de α^i, para $1 \leq i \leq 2t$,

$$g(X) = \mathbf{MCM}\{\Phi_1(X), \Phi_2(X), \ldots, \Phi_{2t}(X)\} \tag{3.81}$$

El grado de cada polinomio mínimo $\Phi_i(X)$ es menor o igual a m, con lo que el grado del polinomio $g(X)$ es como mucho $2mt$. Esto significa que, como mucho habrá $2mt$ elementos de chequeo de paridad. Para el caso de $m = 1$ obtenemos los códigos Reed-Solomon, la subfamilia de códigos BCH $q-arios$ más conocida.

Sea α un elemento primitivo en $GF(q)$, el polinomio generador $g(X)$ de un código RS con símbolos de $GF(q)$ que tiene en $\alpha, \alpha^2, \ldots, \alpha^{2t}$ todas sus raíces queda definido por:

$$\mathbf{g}(X) = (X-\alpha)(X-\alpha^2)\ldots(X-\alpha^{2t}) = g_0+g_1Xg_2X^2+\cdots+g_{2t-1}X^{2t-1}+X^2t \tag{3.82}$$

Ésta está definida bajo el campo $GF(q)$ y respeta los siguientes parámetros

Tamaño del bloque:	$n = q - 1$
número de bits de paridad	$n - k \leq 2t$
Distancia mínima	$d_{min} \geq 2t + 1$
Capacidad correctora	t errores por bloque

Si analizamos las características de los códigos RS, se observa que el tamaño de una palabra código es un símbolo menor que el tamaño del alfabeto. Se observa también que la distancia mínima es un símbolo mayor que el número de símbolos de paridad. Los códigos que obedecen esta última calidad se denominan códigos *de máxima distancia separable*.

La decodificación de un código RS es la misma que la de cualquier código cíclico, obteniéndose mediante la multiplicación polinómica del mensaje con el polinomio generador. Como recordatorio, en forma sistemática, obtenemos los bits de paridad como residuos de la siguiente ecuación:

$$X^{2t}m(X) = q(X)g(X) + p(X) \tag{3.83}$$

3.8.2. Decodificación de los códigos RS

En este punto nos centraremos en el algoritmo de Berlekamp. Es posible también decodificarlos mediante el algoritmo de Euclides, tal como aparece en el apéndice para la decodificación de los códigos BCH binarios. La diferencia para el caso de los códigos RS es que el valor del error no es trivial, y en este caso se tendrá que desarrollar el polinomio evaluador de errores para hallar el valor del error una vez se haya localizado la posición de éste.

3.8.2.1. Algoritmo Berklekamp

En naturaleza, la decodificación de los códigos BCH no binarios, y entre ellos están los RS, es la misma que para el caso binario, con la excepción que ahora no nos basta tan solo con localizar la posición errónea dentro de la secuencia recibida, sino que también debemos calcular su valor.

Sea un código transmitido $m(X)$ y un código recibido $r(X)$,

$$\mathbf{m}(X) = m_0 + m_1 X + \cdots + m_{n-1} X^{n-1} \tag{3.84}$$

$$\mathbf{r}(X) = r_0 + r_1 X + \cdots + r_{n-1} X^{n-1} \tag{3.85}$$

Entonces, cualquier vector de error cumple:

$$\mathbf{e}(X) = \mathbf{r}(x) - \mathbf{m}(X) = e_0 + e_1 X + \cdots + e_{n-1} X^{n-1} \tag{3.86}$$

Donde $\mathbf{X}^{ji}$ son las posiciones de error y $\mathbf{e}_{ji}$ es el valor del error en dicha posición. Por ello, cualquier método de decodificación de un código RS pasará por localizar la posición y el valor del error. Así, el algoritmo que decodifique un código RS consistirá en los siguientes cuatro pasos:

1. Cálculo de los síndromes $(S_1, S_2, ..., S_{2t})$.
2. Cálculo del polinomio localizador de error $\Phi(X)$.
3. Determinar el evaluador de errores
4. Hallar la localización de los errores, el valor de éstos y llevar a cabo la corrección.

A continuación pasaremos a relacionar los síndromes con el valor del error.

$$S_i = r(\alpha^l) = m(\alpha^l) + e(\alpha^l) = e(\alpha^l), \;\; para \;\; 1 \leq l \leq 2t \tag{3.87}$$

Por consiguiente,

$$\begin{aligned}
S_1 &= e_{j1}\alpha^{j1} + e_{j2}\alpha^{j2} + \cdots + e_{jv}\alpha^{jv} \\
S_2 &= e_{j1}(\alpha^{j1})^2 + e_{j2}(\alpha^{j2})^2 + \cdots + e_{jv}(\alpha^{j}v)^2 \\
S_3 &= e_{j1}(\alpha^{j1})^3 + e_{j2}(\alpha^{j2})^3 + \cdots + e_{jv}(\alpha^{j}v)^3 \\
&\vdots \\
S_2t &= e_{j1}(\alpha^{j1})^{2t} + e_{j2}(\alpha^{j2})^{2t} + \cdots + e_{jv}(\alpha^{j}v)^{2t}
\end{aligned} \tag{3.88}$$

Al igual que en los códigos BCH binarios, renombramos la posiciones y, en este caso, el valor del error de la siguiente manera:

$$\beta_l = \alpha^{jl} \quad , \quad \delta_l = e_{jl}$$

$$\begin{aligned} S_1 &= \delta_1\beta_1 + \delta_2\beta_2 + \cdots + \delta_v\beta_v \\ S_2 &= \delta_1\beta_1^2 + \delta_2\beta_2^2 + \cdots + \delta_v\beta_v^2 \\ S_3 &= \delta_1\beta_1^3 + \delta_2\beta_2^3 + \cdots + \delta_v\beta_v^3 \\ &\vdots \\ S_2t &= \delta_1\beta_1^2t + \delta_2\beta_2^{2t} + \cdots + \delta_v\beta_v^{2t} \end{aligned} \tag{3.89}$$

Definimos el polinomio localizador $\sigma(X)$ como:

$$\sigma(X) = (1-\beta_1X)(1-\beta_2X)\ldots(1-\beta_vX) = \sigma_0 + \sigma_1X + \sigma_2X^2 + \cdots + \sigma_vX^v \tag{3.90}$$

Relacionamos los coeficientes σ_l con los componentes del síndrome S_l mediante las *identidades de Newton.*

El algoritmo de Berlekemap consiste básicamente en encontrar las posiciones del polinomio localizador, $\sigma_1, \sigma_2, \ldots, \sigma_v$, cuyas raíces determinan la posición de los errores.

El primer punto del algoritmo consiste en determinar el polinomio de grado mínimo $\sigma^{(1)}$ que satisface la primera identidad de newton. Una vez hallada la ecuación, se testea la segunda identidad de Newton. Si el polinomio $\sigma^{(1)}$ satisface la segunda identidad de Newton entonces $\sigma^{(2)} = \sigma^{(1)}$. De otro modo se añade una corrección en $\sigma^{(1)}$ para formar $\sigma^{(2)}$, capaz de cumplir satisfactoriamente las dos primeras identidades de Newton. Se aplica este procedimiento consecutivamente hasta hallar $\sigma^{(2t)}$. Una vez llegados a este punto, nos quedamos con el polinomio $\sigma^{(2t)}$ como polinomio localizador $\sigma(X), \sigma(X) = \sigma^{(2t)}$.

Existe una forma iterativa de implementar el algoritmo de Berlekamp y obtener el polinomio $\sigma(X)$ en $2t$ pasos. En la iteración $\mu - ava$ se obtiene

el polinomio de grado mínimo que satisface las primeras μ identidades de Newton

$$\sigma^{(\mu)}(X) = \sigma_0^{(\mu)} + \sigma_1^{(\mu)} X + \sigma_2^{(\mu)} X^2 + \cdots + \sigma_{l\mu}^{(\mu)} X^{l\mu} \tag{3.91}$$

El siguiente paso es calcular la igualdad de la siguiente identidad de Newton. Para ello, calculamos la discrepancia d_μ :

$$d_\mu = S_{\mu+1} + \sigma_1^{(\mu)} S_\mu + \cdots + \sigma_{l_\mu}^{(\mu)} S_{\mu+1-l_\mu} \tag{3.92}$$

Si $d_\mu = 0$, entonces:

$$\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X) \tag{3.93}$$

En cambio, si $d\mu \neq 0$ $\sigma^{(\mu)}(X)$ no cumple la $(\mu+1) - ava$ identidad de Newton, por lo que se debe llevar a cabo una corrección de la siguiente forma; volver hacia atrás hasta una iteración anterior ρ tal que la discrepancia $d_\rho \neq 0$ y $\rho - l_\rho$ sea un máximo. El número l_ρ es el grado del polinomio $\sigma^{(\rho)}(X)$.

Entonces,

$$\sigma^(\mu+1)(X) = \sigma^(\mu)(X) + d_\mu {d_\rho}^{-1} X^(\mu-\rho)\sigma^(\rho)(X)$$
$$l_{\mu+1} = max(l_\mu, l_\rho + \mu - \rho)$$

$$d_{\mu+1} = S_{\mu+2} + \sigma^(\mu+1)_1 S_{\mu+1} + \cdots + \sigma_{l_{\mu+1}}^{(\mu+1)} S_{\mu+2-l_\mu} \tag{3.94}$$

Este polinomio de grado mínimo satisface la $(\mu+1) - ava$ identidad de Newton.

Una vez hallado el polinomio localizador, podemos encontrar sus raíces, cosa que se puede hacer de forma sistemática mediante la *búsqueda de Chien*, detallada en el apéndice B.

La siguiente tabla indica el procedimiento para empezar el algoritmo iterativo de Berlekamp.

μ	$\sigma(\mu)(X)$	d_μ	l_μ	$\mu - l_\mu$
-1	1	1	0	-1
0	1	S_1	0	0
1	$1 + S_1 X$			
·				
·				
2t				

Ejemplo 4 *(lin-costello pag 244) Sea el código RS con símbolos de* $GF(2^4)$. *El polinomio generador del código es:*

$$\begin{aligned} \boldsymbol{g}(X) &= (X+\alpha)(X+\alpha^2)(X+\alpha^3)(X+\alpha^4)(X+\alpha^5)(X+\alpha^6) \\ &= \alpha^6 + \alpha^9 X + \alpha^6 X^2 + \alpha^4 X^4 + \alpha^{10} X^5 + X^6 \end{aligned}$$

Supondremos que transmitimos la palabra código todo ceros, y recibimos $r = (000\alpha^7 00\alpha^3 00000\alpha^4 00)$, *que en forma polinómica se corresponde a* $r(X) = \alpha^7 X^3 + \alpha^3 X^6 + \alpha^4 X^{12}$.

El primer paso es calcular los síndromes, obteniendo los valores finales a partir de la tabla A.2 del apéndice.

$$\begin{aligned} S_1 &= \boldsymbol{r}(\alpha) = \alpha^{10} + \alpha^9 + \alpha = \alpha^{12} \\ S_2 &= \boldsymbol{r}(\alpha^2) = \alpha^{13} + 1 + \alpha^{13} = 1 \\ S_3 &= \boldsymbol{r}(\alpha^3) = \alpha + \alpha^6 + \alpha^{10} = \alpha^{14} \\ S_4 &= \boldsymbol{r}(\alpha^4) = \alpha + \alpha^{12} + \alpha^7 = \alpha^{10} \\ S_5 &= \boldsymbol{r}(\alpha^5) = \alpha^7 + \alpha^3 + \alpha^4 = 0 \\ S_6 &= \boldsymbol{r}(\alpha^6) = \alpha^{10} + \alpha^9 + \alpha = \alpha^{12} \end{aligned}$$

A continuación se rellena el cuadro n hasta encontrar $\sigma(X)$.

μ	$\sigma(\mu)(X)$	d_μ	l_μ	$\mu - l_\mu$
-1	*1*	*1*	*0*	*-1*
0	*1*	α^{12}	*0*	*0*
1	$1 + \alpha^{12}X$	*1*	*1*	*1*
2	$1 + \alpha^3 X$	α^7	*2*	*1*
4	$1 + \alpha^4 X + \alpha^1 2X^2$	α^{10}	*2*	*2*
5	$1 + \alpha^7 X + \alpha^4 X^2 + \alpha^6 X^3$	*0*	*3*	*2*
6	$1 + \alpha^7 X + \alpha^4 X^2 + \alpha^6 X^3$			

Iteración $\mu + 1 = 1$*, obtenida a partir de la información de* $\mu = 0$*:*

ρ *debe ser un máximo en un estado anterior a* μ*, por ello es* -1*.*

Calculamos l_μ*:*

$l_{\mu+1} = max(l_\mu, l_\rho + \mu - \rho)$
$l_1 = max(l_0, l_{-1} + 0 - (-1)) = 1$

Calculamos $\sigma(\mu + 1)(X)$*:*

$\sigma^{(\mu+1)}(X) = \sigma^{(\mu)}(X) + d_\mu d_\rho^{-1} X^{(\mu-\rho)} \sigma^{(\rho)(X)}$
$\sigma^{(\mu+1)}(X) = \sigma^{(0)}(X) + d_0 d_{-1}^{-1} X^{(0-(-1))} \sigma^{(-1)}(X)$
$= 1 + \alpha^{12} \cdot 1X \cdot 1 = 1 + \alpha^{12}X$

Calculamos d_1 *:*

$d_{\mu+1} = S_{\mu+2} + \sigma_1^{(\mu+1)} S_{\mu+1} + \cdots + \sigma_{l_{\mu+1}}^{(\mu+1)} S_{\mu+2-l_\mu}$
$d_1 = S_2 + \sigma_1^{(1)} S_1 = 1 + \alpha^{12}\alpha^{12} = 1 + \alpha^{24} = 1 + \alpha^9 = 1 + \alpha + \alpha^3 = \alpha^7$

Iteración $\mu + 1 = 2$*, obtenida a partir de la información de* $\mu = 1$*:*

La distancia es distinta a 0; por ello se ha de hacer una corrección. Así, el primer punto es definir ρ*, en este caso se debe buscar un máximo para* $\rho - l_\rho$ *y* $\rho < 1$*, así* $\rho = 0$*.*

$l_2 = max(l_1, l_0 + 1 - (0)) = 1$
$\sigma^{(\mu+1)}(X) = \sigma^{(1)}(X) + d_1 {d_0}^{-1} X^{(1-0)} \sigma^{(0)}(X)$
$= 1 + \alpha^{12}X + \alpha^7 \alpha^{-12} X \cdot 1 = 1 + \alpha^{12}X + \alpha^{-5}X$
$= 1 + \alpha^{12}X + \alpha^9 X = 1 + \alpha^{12}X + \alpha^{10}X$
$= 1 + (\alpha^{12} + \alpha^{10})X = 1 + \alpha^3 X$

$d_2 = S_3 + \sigma_1^{(2)} S_2 = \alpha^{14} + \alpha^3 = 1 + \alpha^3 + \alpha^3 = 1$

Iteración $\mu + 1 = 3$*, obtenida a partir de la información de* $\mu = 2$*:*

$\rho = 0$
$l_3 = max(l_2, l_0 + 2 - 0) = max(1, 0 + 2) = 2$
$\sigma^{(\mu+1)}(X) = \sigma^{(3)}(X) = \sigma^{(2)}(X) + d_2 {d_0}^{-1} X^{(2-0)} \sigma^{(0)}(X)$
$= 1 + \alpha^3 X + 1 \cdot \alpha^{-12} X^2 \cdot 1 = 1 + \alpha^3 X + \alpha^3 X^2$
$d_3 = S_4 + \sigma_1^{(3)} S_3 + \sigma_2^{(3)} S_2$
$= \alpha^{10} + \alpha^3 \cdot \alpha^{14} + \alpha^3 \cdot 1 = \alpha^{10} + \alpha^2 + \alpha^3 = \alpha^7$

Iteración $\mu + 1 = 4$*, obtenida a partir de la información de* $\mu = 3$*:*

$\rho = 2$
$l_4 = max(l_3, l_2 + 3 - 2) = max(2, 1 + 3 - 2) = 2$
$\sigma^{(\mu+1)}(X) = \sigma^{(4)}(X) = \sigma^{(3)}(X) + d_3 {d_2}^{-1} X^{(3-2)} \sigma^{(2)}(X)$
$= 1 + \alpha^3 X + \alpha^3 X^2 + \alpha^7 \cdot 1 \cdot X(1 + \alpha^3 X)$
$= 1 + \alpha^4 X + \alpha^{12} X^2 d_4 = S_5 + \sigma_1^{(4)} S_4 + \sigma_2^{(4)} S_3$
$= 0 + \alpha^4 \cdot \alpha^{10} + \alpha^{12} \cdot \alpha^{14} = \alpha^{14} + \alpha^{11} = \alpha^{10}$

Iteración $\mu + 1 = 5$*, obtenida a partir de la información de* $\mu = 4$*:*

$\rho = 3$
$l_5 = max(l_4, l_3 + 4 - 2) = max(2, 2 + 4 - 3) = 3$
$\sigma^{(\mu+1)}(X) = \sigma^{(5)}(X) = \sigma^{(4)}(X) + d_4 {d_3}^{-1} X^{(4-3)} \sigma^{(3)}(X)$
$= 1 + \alpha^4 X + \alpha^{12} X^2 + \alpha^{10} \cdot \alpha^{-7} \cdot X(1 + \alpha^3 X + \alpha^3 X^2)$
$= 1 + \alpha^7 X + \alpha^4 X^2 + \alpha^6 X^3$
$d_5 = S_6 + \sigma_1^{(5)} S_5 + \sigma_2^{(5)} S_4 + \sigma_3^{(5)} S_3$
$= \alpha^{12} + \alpha^7 \cdot 0 + \alpha^4 \cdot \alpha^{10} + \alpha^6 \cdot \alpha^{14}$
$= \alpha^{12} + \alpha^{14} + \alpha^5 = 0$

En este caso, la distancia es $d_\mu = 0$*. Así,* $\sigma(\mu + 1)(X) = \sigma^{(\mu)}(X)$*, por lo que no nos hace falta calcular el siguiente y último polinomio localizador.*

$$\sigma(X) = 1 + \alpha^7 X + \alpha^4 X^2 + \alpha^6 X^3$$

A continuación se deben encontrar las raíces del polinomio localizador, cosa que se puede hacer mediante la búsqueda de Chien:

$\sigma(0) = 1+\alpha^7+\alpha^4+\alpha^6 = 1+1+\alpha+\alpha^3+1+\alpha+\alpha^2+\alpha^3 = 1+\alpha^2 = \alpha^8 \neq 0$
$\sigma(\alpha) = 1+\alpha^8+\alpha^5+\alpha^7 = \alpha^{14} \neq 0$
$\sigma(\alpha^2) = 1+\alpha^9+\alpha^8+\alpha^{12} = 1 \neq 0$
$\sigma(\alpha^3) = 1+\alpha^{10}+\alpha^{10}+\alpha^{15} = 0$

$\vdots$

$\sigma(\alpha^9) = 1+\alpha^{16}+\alpha^{22}+\alpha^{33} = 1+\alpha+\alpha^7+\alpha^3 = 0$

$\vdots$

$\sigma(\alpha^{12}) = 1+\alpha^{19}+\alpha^{28}+\alpha^{42} = 1+\alpha^4+\alpha^{13}+\alpha^{12} = 0$

$\vdots$

Tres raíces son encontradas mediante la búsqueda de Chien , por lo que la posiciones de los errores α^h son: $\alpha^3, \alpha^9, \alpha^{12}$. Por lo que las posiciones r_{n-h} son: r_{12}, r_6 y r_3.

Una vez halladas las posiciones, se deben determinar los valores de error. A diferencia del caso binario, donde siempre tenían el valor 1 (véase ejemplo apéndice n), en este caso su valor no es trivial. Para hallar estos valores definimos las siguientes funciones:

$$\boldsymbol{Z}_0(X) = \sum_{l=1}^{v} \delta_l \beta_l \prod_{i=1, i\neq l}^{v} (1-\beta_i) \tag{3.95}$$

Mediante métodos algebraicos se puede relacionar la función anterior con los síndromes y los coeficientes del polinomio evaluador:

$$\boldsymbol{Z}_0 = S_1+(S_2+\sigma_1 S_1)X+(S_3+\sigma_1 S_2+\sigma_2 S_1)X^2+\cdots+(S_v+\sigma_1 S_{v-1}+\cdots+\sigma_{v-1}S_1)X^{v-1} \tag{3.96}$$

Así, el valor del error se puede obtener mediante la siguiente expresión:

$$\delta_k = \frac{-\boldsymbol{Z}_0(\beta_k^{-1})}{\sigma'(\beta_k{}^{-1})} \tag{3.97}$$

Otra forma de definir el polinomio evaluador es la siguiente:

$$\begin{aligned}\boldsymbol{Z}(X) &= \sigma(\mathbf{X}) + X\,\boldsymbol{Z}_0(X) \\ &= 1 + (S_1 + \sigma_1 S_1)X + (S_2 + \sigma_1 S_1 + \sigma_2)X^2 + \cdots + (S_\upsilon + \sigma_1 S_{\upsilon-1} + \cdots + \sigma_\upsilon)X^\upsilon\end{aligned} \tag{3.98}$$

y el valor del error como:

$$\delta_k = \frac{\text{-}\boldsymbol{Z}(\beta_k^{-1})}{\prod_{i=1,i\neq k}^{\upsilon}(1-\beta_i\beta_k{}^{-1})} \tag{3.99}$$

Continuando con el ejemplo, ahora ya estamos en disposición de determinar el valor de los errores. Mediante el algoritmo de Berlekamp encontramos el polinomio evaluador:

$$\sigma(X) = 1 + \alpha^7 X + \alpha^4 X^2 + \alpha^6 X^3$$

Cuyas posiciones de error están en X^3, X^6, X^{12}. El primer paso para hallar el valor de error en estas posiciones es definiendo la función Z_0,

$$\begin{aligned}Z_0 &= S_1 + (S_2 + \sigma_1 S_1)X + (S_3 + \sigma_1 S_2 + \sigma_2 S_1)X^2 \\ &= \alpha^{12} + (1 + \alpha^7\alpha^{12})X + (\alpha^{14} + \alpha^7 + \alpha^4\alpha^{12})X^2 \\ &= \alpha^{12} + (1 + \alpha^4)X + (\alpha^{14} + \alpha^7 + \alpha)X^2 = \alpha^{12} + \alpha X\end{aligned}$$

$$\sigma'(X) = \alpha^7 + 2\alpha^4 X + 3\alpha^6 X^2 = \alpha^7 + \alpha^6 X^2$$

$$e_3 = \frac{-\boldsymbol{Z}_0(\alpha^{-3})}{\sigma'(\alpha^{-3})} = \frac{\alpha^{12}+\alpha^{13}}{\alpha^7+1} = \frac{\alpha}{\alpha^9} = \alpha^7$$

$$e_6 = \frac{-\boldsymbol{Z}_0(\alpha^{-6})}{\sigma'(\alpha^{-6})} = \frac{\alpha^{12}+\alpha^{10}}{\alpha^7+\alpha^9} = \frac{\alpha^3}{1} = \alpha^3$$

$$e_12 = \frac{-\boldsymbol{Z}_0(\alpha^{-12})}{\sigma'(\alpha^{-12})} = \frac{\alpha^{12}+\alpha^{10}}{\alpha^7+\alpha^{12}} = \frac{\alpha^6}{\alpha^2} = \alpha^3$$

De modo que el vector del error es el siguiente:

$\boldsymbol{e}(X) = \alpha^7 X^3 + \alpha^3 X^6 + \alpha^4 X^{12}$

que es exactamente la secuencia recibida, de modo que:

$\boldsymbol{e}(X) = \boldsymbol{r}(X), \boldsymbol{m}(X) = \boldsymbol{e}(X) + \boldsymbol{r}(X) = \boldsymbol{0}$

Capítulo 4

Códigos convolucionales

Un codificador (polinomio) convolucional es una aplicación inyectiva de F(z) módulos:

$$F(Z)^k \xrightarrow{G} F(Z)^n$$

El alfabeto de nuestro código está sacado de un campo finito o cuerpo de galois de modo que tiene unas características especiales que quedan reflejadas en el anexo A.

La aplicación G va a venir dada por una matriz formada por polinomios $g_i(z)$. Si la matriz tiene polinomios constantes tenemos un código lineal de bloques y no un codificador convolucional. Un código convolucional es la imagen de $G\text{Im}\ \{G\}$.

En los códigos convolucionales la codificación en cada instante de tiempo depende de como hemos codificado en varios instantes anteriores. El número de pasos anteriores de los cuales dependemos es lo que se llama el grado del polinomio convolucional o memoria del código convolucional. Un código lineal de bloques es un codificador convolucional de grado 0, o sea, sin memoria; a igual palabra de mensaje, igual codificación. No es así en los códigos convolucionales.

Un codificador convolucional coge una $k - upla\ m_i$, como mensaje de entrada y genera una $n - upla\ c_i$ de elementos codificados como salida en un

instante i. La codificación en este instante, como hemos dicho anteriormente, no depende solo de la $k - upla$ m_i sino también de $k - uplas$ previas m_j, donde $j < i$.

Veamos el esquema de un codificador convolucional,

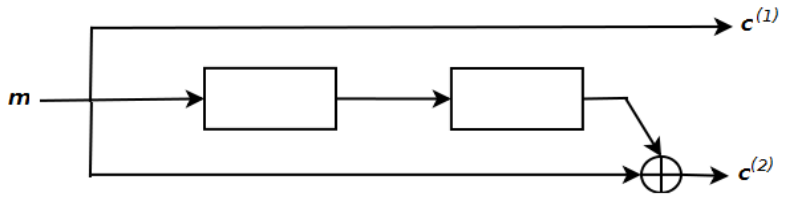


Figura 4.1: *Esquema de un codificador convolucional sistemático con dos estados de memoria*

Como vemos, no es más que una máquina de estados finita,FSSM (*finite state sequential machine*).

El codificador convolucional de la figura 4.1 es sistemático, ya que una de sus salidas es la entrada c1. Por la otra salida obtenemos el bit codificado o bit de paridad. Los bloques representan los retardos o unidades de memoria (equivalentes a la duración de un elemento del cuerpo de galois $GF(q)$). Este codificador tiene, por lo tanto, memoria 2. Los círculos representan las sumas dentro del cuerpo de galois.

La salida de este codificador se puede definir como la convolución entre la entrada y la respuesta impulsional del codificador para cada salida n. Como es conocido, la respuesta impulsional se obtiene aplicando el impulso unidad a la entrada.

Destacaremos los siguientes parámetros:

- $k =$ *tasa de entrada*
- $n =$ *tasa de salida*
- $K =$ *memoria del código*
  De estos parámetros también obtenemos:

- $K+1 = constraintlength$, el número máximo de unidades de tiempo que dado un bit de la secuencia de entrada puede influenciar en la secuencia de salida.
- $r = Tasa\ del\ codificador\ = \frac{n}{k}$, relación entre los bits que entran y los que salen del codificador.

En el codificador convolucional de la figura 4.1 destacamos los siguientes parámetros:

- *tasa de entrada* $k = 1$
- *memoria del codificador* $K = 2$
- *tasa de salida* $n = 2$.
- *constraint lengh* $t = 3$
- *tasa del codificador* $r = 1/2$.

Como la *constraintlenght* es 3, el mensaje unidad que nos calcula la respuesta impulsional es $m = (100)$, de longitud 3. Veamos en una tabla como queda la salida:

i	m	S_1	S_2	c_1	c_2
0	1	0	0	1	1
1	0	1	0	0	0
2	0	0	1	0	1
3	0	0	0	0	0

Como podemos comprobar, el bit 1 nos influye desde el estado 0 al estado 2, o sea, tres estados, que es la constraint lenght. Ahora, si nos quedamos con la salida a la entrada impulso obtenemos la respuesta impulsional:

$$c_1 = g_1 = (100)$$
$$c_2 = g_2 = (101)$$

La respuesta impusional g_1 y g_2 nos define las conexiones del codificador, habiendo adiciones allí donde hay unos.

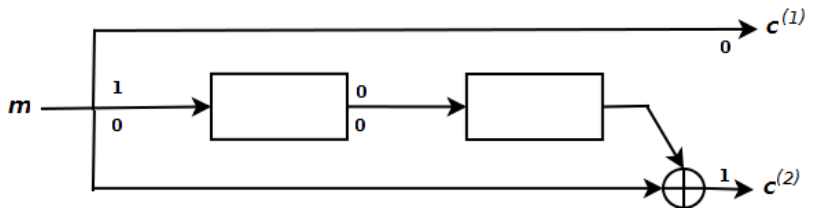


Figura 4.2: *Esquema de un codificador convolucional sistemático con dos estados de memoria*

La respuesta impusional g nos define la aplicación G que representa el codificador, de modo que para obtener la salida tan solo hemos de hacer una convolución

$$c_1 = u * g_1 c_2 = u * g_2 \tag{4.1}$$

En términos generales tenemos:

$$c_l^{(j)} = \sum_{i=0}^{k} m_{l-i} g_i^{(j)} = m_l g_0^{(j)} + m_{l-1} g_1^{(j)} + \cdots + m_{l-k} g_k^{(j)} \tag{4.2}$$

En el caso de nuestro ejemplo en particular,

$$c_l^1 = \sum_{i=0}^{2} m_{l-i} g_i^{(1)} = m_l \tag{4.3}$$

$$c_l^2 = \sum_{i=0}^{2} m_{l-i} g_i^{(2)} = m_l + m_{l-2} \tag{4.4}$$

4.1. Representación Polinómica

Como sabemos, todas aquellas convoluciones temporales tienen su equivalente en una multiplicación en el dominio transformado. La transformada D

nos permite expresar los mensajes en forma polinómica, siendo D los retardos temporales de nuestro codificador.

$$m^l = (m_0^l, m_1^l, m_2^l, \dots) \rightarrow M^l(D) = m_0^l + m_1^l D + m_2^l D^2 + \dots \tag{4.5}$$

donde los números en los subíndices representan los espacios temporales del bit de entrada o salida, i equivale a la entrada i del codificador, siendo j el número de salida.

De este modo, el mensaje codificado se obtiene de la forma:

$$C^1(D) = M(D)G^1(D) = M(D) \tag{4.6}$$

$$C^2(D) = M(D)G^2(D) = M(D)(1 + D^2) \tag{4.7}$$

La memoria de este codificador es el grado máximo del polinomio, que es 2. En el caso de más de una entrada y salida, la memoria de cada entrada i es el máximo de la memoria que tiene cada camino de dicha entrada.

$$K_i = \underset{1 \leq i \geq n}{max} \left\{ deg(g_i^j D) \right\} \quad , 1 \leq i \leq k \tag{4.8}$$

La memoria del codificador, K, se obtiene del máximo de las memorias de cada entrada.

$$K = \underset{1 \leq i \leq k}{max} K_i = \underset{1 \leq i \leq n, 1 \leq i \leq k}{max} \left\{ deg(g_i^j D) \right\} \tag{4.9}$$

En el caso de tener varias entradas y salidas, si representamos el mensaje en la entrada i mediante M_i y la salida j a este mensaje de entrada mediante C_j, la función de transferencia que nos relaciona sendos mensajes es:

$$G_i^{(j)} = \frac{C^(j)(D)}{M^(i)(D)} \tag{4.10}$$

En el caso más general de k entradas y n salidas, habrá kn funciones de transferencia que quedaran reflejadas mediante una matriz de la forma siguiente:

$$\begin{bmatrix} G_1^{(1)}(D) & G_1^{(1)}(D) & \dots & G_1^{(n)}(D) \\ G_2^{(1)}(D) & G_2^{(2)}(D) & \dots & G_2^{(n)}(D) \\ \vdots & \vdots & & \vdots \\ G_k^{(1)}(D) & G_k^{(2)}(D) & \dots & G_k^{(n)}(D) \end{bmatrix} \tag{4.11}$$

De este modo :

$$C(D) = M(D) * G(D) \tag{4.12}$$

si $M(D) = (M^{(1)}(D), M^{(2)}(D), M^{(3)}(D), \dots, M^{(4)}(D))$, y
$C(D) = (C^{(1)}(D), C^{(2)}(D), \dots, C^{(3)}(D))$

Multiplexando, nos queda:

$$C_m(D) = C^{(1)}(D^n) + DC^{(2)}(D^n) + D^2C^{(3)}(D^n) + \cdots + D^{n-1}C^{(n)}(D^n) \tag{4.13}$$

4.2. Representación de los códigos convolucionales

4.2.1. Diagrama de estados.

Como hemos dicho, en (k) los códigos convolucionales se pueden ver como una máquina de estados finita compuesta por retardos y adiciones. El conexionado de esta FSSM vendría dado por los 1 y 0 de G.

- En la máquina de estados que representa un codificador convolucional, el contenido de los registros representa los estados.
- La salida del codificador en un instante de tiempo t depende de la entrada en t y del estado actual .

- Cada cambio de estado va asociada a una secuencia de información a la entrada y a una codificación a la salida.

- No es posible pasar de un estado a otro de forma aleatoria. Por ello, esto representa un tipo de memoria, ya que si estoy en un estado en instante de tiempo, puedo saber los posibles estados anteriores o bien deducir que la secuencia recibida no es válida.

Ejemplo 5 *Supongamos la secuencia de entrada u=01010111000, si trabajamos con el codificador utilizado en el ejemplo, obtenemos la siguiente tabla de posibles transiciones:*

entrada m_i	*Estado a* t_i	*Estado a* t_{i+1}	$c^{(1)}$	$c^{(2)}$
0	*00*	*00*	*0*	*0*
1	*00*	*10*	*1*	*1*
0	*10*	*01*	*0*	*0*
1	*01*	*10*	*1*	*0*
0	*10*	*01*	*0*	*0*
1	*01*	*10*	*1*	*0*
1	*10*	*11*	*1*	*1*
1	*11*	*11*	*1*	*0*
0	*11*	*01*	*0*	*1*
0	*01*	*00*	*0*	*1*
0	*00*	*00*	*0*	*0*

En la tabla anterior se muestran todas las posibles transiciones entre estados así como sus respectivas secuencias codificadas. Como podemos observar, una vez estamos en el estado 1 1 necesitamos tres secuencias seguidas de 0, para que la codificación de salida sea distinta a la anterior, siempre se codificará 0 0 y siempre estaremos en el mismo estado, es decir a partir de $K+1$ *ceros la codificación se hace repetitiva. Eso es lo que llamamos constraint length. Si realizamos una representación gráfica de las transiciones entre estado y las secuencias codificadas tenemos el siguiente diagrama de estados:*

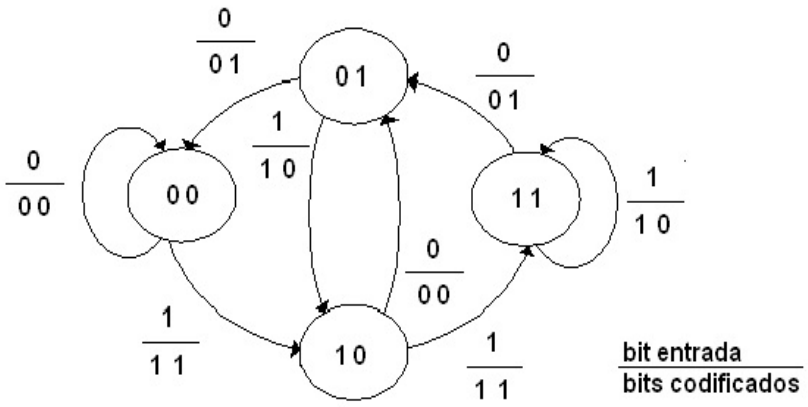


Figura 4.3: *Diagrama de estados del codificador convolucional sistemático de la figura 4.2*

En este tipo de representación se puede ver de forma clara la relación entre estados y secuencias de entrada y salida, No obstante, por otro lado hay una variable que no está representada en este tipo de gráficas y que es de suma importancia para los codificadores con memoria e iterativos: el tiempo. En un *Diagrama de Estados* no sabemos cuando pasamos de **a** a **b** ni por qué estados hemos pasado anteriormente. Para ello, necesitamos otra representación, el diagrama de trellis o enrejado.

4.2.2. Diagrama de trellis

Un diagrama de trellis se puede ver como la evolución de un diagrama de estados a través del tiempo, o un diagrama de estados al que se le ha añadido la variable temporal.

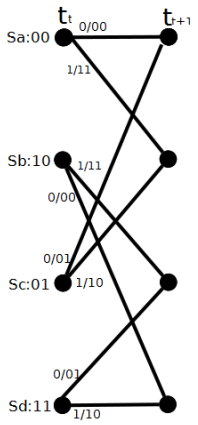


Figura 4.4: *Diagrama de estados del codificador convolucional sistemático de la figura 4.2*

Veamos como nos queda representada la secuencia anterior $u = 01010111000$, y el codificador del ejemplo mediante un diagrama de Trellis:

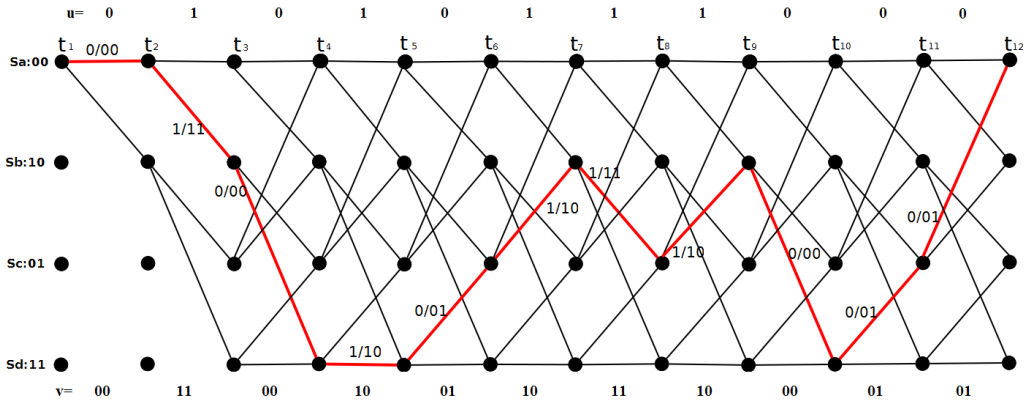


Figura 4.5: *Diagrama de trellis del ejemplo*

En este tipo de diagramas observamos la evolución temporal del código, y que para un mismo bit de entrada no tenemos una única salida, sino que dependemos del estado en que nos encontremos. Así, la secuencia de salida viene dada por las transiciones entre estados o caminos del trellis, de modo que una codificación convolucional tiene asociado un camino dentro del

diagrama de trellis de forma biunívoca. En el caso de nuestra secuencia de entrada $u = 01010111000$, el camino dentro del trellis según el codificador de la figura 4.2 es el representado de color rojo. Las lineas negras más finas son las posibles transiciones entre estados.

4.3. Propiedades de distancia de un código convolucional

Recordemos que la distancia o distancia de hamming entre dos secuencias código x y es el número de posiciones en las que difiere el uno del otro. La distancia mínima de un código convolucional sería el número mínimo de errores sobre una palabra código que nos llevará a otra palabra código válida.

4.3.1. Mínima distancia libre de un código convolucional.

La mínima distancia libre de un código convolucional queda definido por:

$$d_{free} = min\{d(c_i, c_j) : m_i \neq m_j\} \tag{4.14}$$

donde c_i y c_j son dos secuencias código correspondiente al los mensajes m_i y m_j. Se asume que c_i y c_j empiezan y acaban en la secuencia todo ceros y que tienen longitud finita. Si aplicamos las propiedades de linealidad podemos representar la mínima distancia libre de un código convolucional como el peso de hamming del código, es decir:

$$d_{free} = min\{(c_i + c_j) : m_i \neq m_j\} = min\{w(c) : m \neq 0\} \tag{4.15}$$

La última igualdad se obtiene de la propiedad de linealidad por la cual de la suma de dos palabras código se obtiene otra palabra código. De este modo, la mínima distancia libre no es más que el mínimo peso de hamming de todas las secuencias del código que empiezan y terminan el estado todo ceros y que son distintas a cero.

4.3.2. Función de distancia de columna (CDF)

En el caso de los códigos convolucionales, la distancia entre dos secuencias código no es tan clara como en el caso de los códigos bloques, ya que en los códigos convolucionales no tenemos definido el tamaño de nuestra secuencia y la memoria del código juega un importante papel. Así, en este tipo de codificaciones se ha de considerar otra función de medida fundamental, la *distancia de columna*. Si consideramos $[v]_j = (v_0^{(0)}, v_0^{(1)}, v_0^{(n-1)}, v_1^{(0)}, v_1^{(1)}, \ldots, v_1^{(n-1)}, \ldots, v_j^{(0)}, v_j^{(1)}, \ldots, v_j^{(n-1)})$ como la *truncación* $j-ava$ de la de la palabra código v, y $[m]_j = (m_0^{(1)}, m_0^{(2)}, \ldots, m_0^{(k)}, m_1^{(1)}, m_1^{(2)}, \ldots m_1^{(k)}, \ldots, m_j^{(1)}, m_j^{(2)}, \ldots, m_j^{(k)})$ como la *truncación* $j-ava$ de la secuencia de información m, entonces se define la CDF de orden j, d_j, como:

$$d_j = min\left\{d([v']_j, [v'']_j) : [m']_0 \neq [m'']_0\right\} = min\left\{w[v]_j : [m]_0 \neq 0\right\} \quad (4.16)$$

donde, en este caso, siguiendo la notación de LIN[04], v' y v'' representan la codificación sobre los mensajes de información m' y m''. De la definición obtenemos que d_j es la mínima palabra código sobre las primeras $(j+1)$ unidades de tiempo de entre las palabras código producidas por secuencias de información cuyo bloque inicial es distinto a cero Dos casos tienen especial interés: cuando $j = K$ y $j \to \infty$; y cuando $j = K$,grado de memoria del codificador convolucional, d_K se denota como la mínima distancia del código convolucional, d_{min}. Así d_{min} representa el mínimo peso de las palabras código sobre las primeras $(K+1)$ unidades de tiempo de entre las palabras código producidas por secuencias de información cuyo bloque inicial es distinto a cero. En el caso de códigos con memoria infinita se obtiene la siguiente definición:

$$\lim_{j\to\infty} d_j = d_{free} \quad (4.17)$$

Anotar que esta definición no es válida para los convolucionadores catastróficos, es decir, aquellos que tienen una respuesta finita a una entrada infinita

De todos modos, el computo de este límite es claramente impracticable computacionalmente. Afortunadamente, existe un limite j_0 para el cual $d_j = d_{free}$ $j \geq j_0$, y esta j_0 es la constraint lenght $(K+1)$.

La capacidad correctora del código queda definida por:

$$t = \left| \frac{d_{free}-1}{2} \right| \tag{4.18}$$

Esta capacidad de error se obtiene cuando los eventos de error están separados, por lo menos, por la constraint length del código medido en bits.

4.4. Decodificación óptima de los códigos convolucionales

4.4.1. Algoritmo de Viterbi

El procedimiento de decodificación del algoritmo de Viterbi consiste en calcular la distancia acumulada entre la secuencia recibida en un instante t_i en un estado del trellis y cada una de todas las posibles secuencias que llegan a ese mismo estado en el instante t_i. El algoritmo de Viterbi reduce la complejidad del cálculo evitando el hecho de tener que anotar todas las posibles secuencias del trellis. Este cálculo se hace para todos los estados del trellis y durante los sucesivos instantes de tiempo con el objetivo de hallar el camino con la mínima distancia acumulada. Una vez hallado, ésta secuencia es la que tiene la probabilidad más alta de ser la secuencia transmitida si esta se ha hecho a través de un canal $AWGN$. Para empezar a desgranar este algoritmo, recordaremos las características del diagrama de trellis.

Para un codificador (n, k, K) y un mensaje de longitud L, un diagrama de trellis contiene $L+K$ unidades de tiempo o niveles, que van del instante $t = 0$ al $t = K + L$. Para una secuencia de tamaño kL, en el diagrama de trellis existirán 2^k ramas saliendo y entrando de cada estado y 2^{kL} trayectorias distintas a través del trellis, las cuales representan las 2^{kL} palabras posibles del codificador convolucional, cada una de las cuales con una longitud $N = n(K + L)$ bits de información.

Otro concepto que debe quedar claro es el concepto del *criterio de máxima verosimilitud*, pues el algoritmo de Viterbi implementa una decodificación de máxima verosimilitud.

Si asumimos que una secuencia de información $m = (m, \ldots, m_{L-1})$ de longitud L se codifica en una palabra código $v = (v_0, v_1, \ldots, v_{L+K-1})$ de longitud N= (L+K) y que una secuencia $Q - aria$ $r = (r_0, r_1, \ldots, r_{L+K-1})$ es la secuencia recibida a través de un canal de entrada binaria y salida $Q - aria$, discreto y sin memoria (DMC), el decodificador producirá una estimación v' de la palabra código $\mathbf{v}$ basándose en la secuencia recibida $\mathbf{r}$. Un decodificador de máxima verosimilitud elige $\mathbf{v}'$ como la palabra código que maximiza la función log-likelihood dada por:

$$log(P(\frac{r}{v})) = \sum_{i=0}^{L+K-1} logP(\frac{\mathbf{r}_i}{\mathbf{v}_i}) = \sum_{i=0}^{N-1} logP(\frac{r_i}{v_i}) \tag{4.19}$$

donde $P(\frac{r_i}{v_i})$ es la probabilidad que el símbolo r_i sea recibido a la salida del canal sin memoria si el símbolo transmitido a través de dicho canal fue v_i.

La función $log - likelihood$ se conoce como la métrica asociada con el camino (palabra código) v, y se denota $M(\frac{r}{v})$. Los términos $logP(\frac{\mathbf{r}_i}{\mathbf{v}_i})$ en los sumandos de la ecuación anterior se denominan métricas *de rama* (branch metrics) y son denotas por $M(\frac{\mathbf{r}_i}{\mathbf{v}_i})$, mientras que los términos $logP(\frac{\mathbf{r}_i}{\mathbf{v}_i})$ son conocidos como métricas de *bit* (bit metrics) y se denotan por $M(\frac{\mathbf{r}_i}{\mathbf{v}_i})$. De ahí podemos escribir la métrica del camino por:

$$M(\frac{\mathbf{r}}{\mathbf{v}}) = \sum_{i=0}^{L+K-1} M(\frac{\mathbf{r}_i}{\mathbf{v}_i}) = \sum_{i=0}^{L+K-1} logP(\frac{\mathbf{r}_i}{\mathbf{v}_i}) = \sum_{i=0}^{N-1} M(\frac{r_i}{v_i}) = \sum_{i=0}^{N-1} logP(\frac{r_i}{v_i}) \tag{4.20}$$

Ahora podemos expresar una métrica parcial para las primeras j ramas del camino como

$$M(\left[\frac{\mathbf{r}}{\mathbf{v}}\right]) = \sum_{i=0}^{j-1} M(\frac{\mathbf{r}_i}{\mathbf{v}_i}) = \sum_{i=0}^{j-1} logP(\frac{\mathbf{r}_i}{\mathbf{v}_i}) = \sum_{i=0}^{nj-1} M(\frac{r_i}{v_i}) = \sum_{i=0}^{nj-1} logP(\frac{r_i}{v_i}) \tag{4.21}$$

El siguiente algoritmo encuentra sobre una secuencia recibida r proveniente de un canal DMC el camino a través del trellis con la métrica más

grande, es decir, el camino más *verosímil*. El algoritmo se procesa de forma recursiva guardando en cada estado la métrica de cada camino que converge a dicho estado. Así, se descartan aquellos caminos que, convergiendo en el mismo estado al mismo tiempo, tienen peores métricas:

Algoritmo de Viterbi

1. Empezando por la unidad de tiempo K, se calcula la métrica parcial del camino que llega a cada estado en ese instante de tiempo partiendo todos los caminos del estado 0. El camino que llega a cada estado antes del instante de tiempo K será único si todos los caminos parten de 0. Estos caminos se denominarán *supervivientes*.

2. Se incrementa t en una unidad. Se calcula la métrica para todos los $2k$ caminos que entran en un estado, añadiendo la métrica de los caminos supervivientes anteriores. Una vez calculados todos caminos, descartamos aquellos con las métricas más bajas y nos quedamos con los caminos con las métricas mayores. Estos pasarán a ser lo caminos supervivientes.

3. Si $t < L + K$, repetimos el paso 2; de otro modo, paramos.

Desde el punto de vista de la implementación, es más conveniente utilizar métricas enteras que métricas de bit. Por ello, la métrica de bit $M(\frac{r_i}{v_i}) = logP(\frac{r_i}{v_i})$ se puede reemplazar por $c_2[logP(\frac{r_i}{v_i}) + c_1]$, donde c1 es cualquier número real y c_2 es cualquier número real positivo. Así, se puede demostrar:

$$max\left[M(\frac{r}{v}) = \sum_{i=0}^{N-1} logP(\frac{r_i}{v_i})\right] = max\left[\sum_{i=0}^{N-1} c_2[logP(\frac{r_i}{v_i}) + c_2\right] \qquad (4.22)$$

De modo que podemos utilizar las métricas modificadas sin alterar el resultado del algoritmo de Viterbi.

En el caso especial de un canal binario simétrico (BSC) con probabilidad de transición $p < \frac{1}{2}$, la secuencia recibida r es binaria y la función $log - likelihood$ pasa a ser:

$$logP(\frac{\mathbf{r}}{\mathbf{v}}) = d(\mathbf{r},\mathbf{v})log\frac{p}{1-p} + Nlog(1-p) \tag{4.23}$$

donde $d(r,v)$ es la distancia de Hamming entre r y v. En este caso el algoritmo de máxima verosimilitud nos reporta el camino con la mínima distancia de Hamming:

$$d(\mathbf{r},\mathbf{v}) = \sum_{K+L-1}^{i=0} d(\mathbf{r}_i,\mathbf{v}_i) = \sum_{i=0}^{N-1} d(r_i,v_i) \tag{4.24}$$

La aplicación del algoritmo de viterbi es exactamente igual, la única variación es que ahora la distancia de hamming reemplaza a la función de máxima verosimilitud como métrica, y el camino superviviente es aquel que tiene la métrica más pequeña.

Ejemplo 6 *Ejemplo de decodificación de viterbi sobre un canal BSC. Siguiendo con el codificador del ejemplo, tenemos un mensaje* $m = 010101110$ *que, una vez se la añaden los bits de cola para asegurar que termina en el estado* 0*, queda un mensaje de longitud* $L' = K + L = 11$*,* $m' = 01010111000$ *con una señal codificada* $v = 00110010011011100000101$*, la cual, después de pasar por el decodificador, queda* $r = 00100010000101110000101$*.*

Primero de todo, calculamos la métrica hasta el estado de tiempo $t = K = 2$ *partiendo del estado* 0*.*

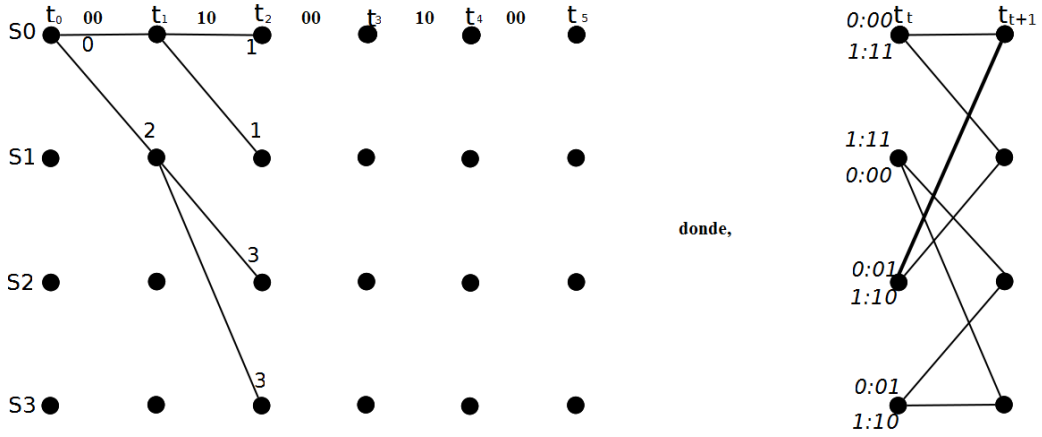


Figura 4.6: *Ejemplo de decodificación de viterbi sobre un canal BSC*

Calculamos la métrica en los instantes sucesivos, cuando en un nodo converjan dos caminos. Descartaremos aquéllos con la distancia de hamming mayor.

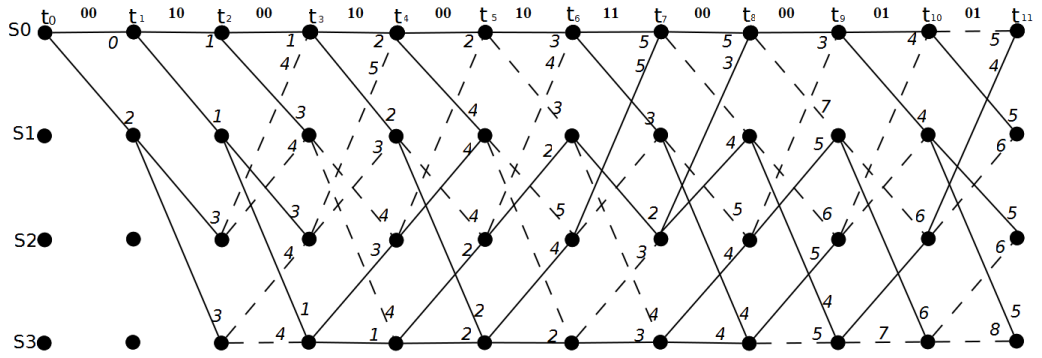


Figura 4.7: *Ejemplo de decodificación de viterbi sobre un canal BSC*

Finalmente llegamos al estado $t = L+K = 11$ *observamos que la métrica con menor distancia de hamming es* $d(r, v) = 4$. *El paso final es reconstruir el camino inverso desde* $t_1 1$ *hasta* t_0. *La secuencia decodificada será aquella que se corresponda con el camino superviviente después de aplicar el algoritmo de viterbi.*

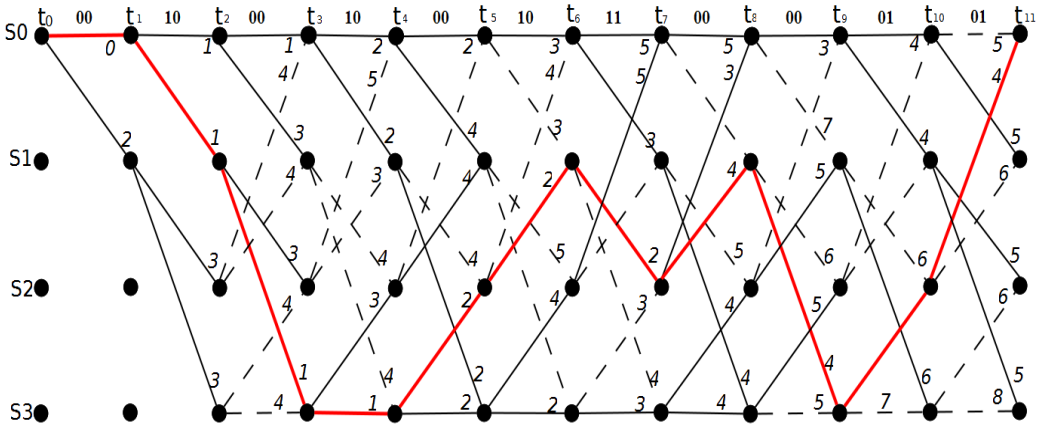


Figura 4.8: *Ejemplo de decodificación de viterbi sobre un canal BSC*

Observamos que el camino resultante es $S_0S_0S_1S_3S_3S_2S_1S_2S_1S_3S_2S_0$ *y este camino, según el trellis de nuestro codificador (figura 4.4), se corresponde con la secuencia codificada* $m' = v = 00110010011011110000101$.

Observaciones:

En este caso, la capacidad correctora del código ha sido suficiente para corregir los dos errores que le hemos introducido sin encontrarnos ninguna situación de dualidad. Por otro, lado si nos fijamos en el instante de tiempo $t = 8$*, en el estado* S_3*, hay dos caminos que convergen en él con la misma distancia acumulada. En este caso, se ha de elegir un camino superviviente de entre los dos de forma aleatoria. Aún así, si la capacidad correctora del código es mayor que los errores introducidos por el canal, no pasaremos por este estado. En caso contrario, el decodificador fallaría y, normalmente, la salida se vería afectada por una ráfaga de errores.*

4.4.2. Matlab y los códigos convolucionales.

Aprovecharemos este apartado sobre los códigos convolucionales para mostrar como se ha utilizado la herramienta matemática matlab para crear la codificación convolucional.

- Función convenc

La sintaxis para la función convenc es la siguiente:

[codigo **estadofinal**]=convenc(***mensaje, trellis***)

Observamos que tenemos cuatro variables de importancia; mensaje, trellis, código y estadofinal. Naturalmente, el código es el mensaje codificado según la estructura de trellis dada por **trellis** y devuelta en la variable **codigo**, **estadofinal** es el estado del trellis donde nos quedamos al terminar la codificación del mensaje.

- Función poly2trellis

trellis = poly2trellis(*ConstraintLength*,*codegenerator*,*FeedbackConnection*)

Como vemos las variables son *constraintlength* , código generador y *FeedbackConnection*, que se utiliza para generar un codificador convolucional sistemático, y su salida es una variable de tipo estructura que representa el trellis del codificador convolucional.

Veamos con más detalle cada una de estas variables:

- *ConstraintLength* es un vector donde cada elemento es el constraint length del codificador. En nuestro caso es uno, pero para el caso de más de una entrada, cada una de ellas tendrá su propia constraint length correspondiente, como se ha visto anteriormente, al número de retardadores más uno.

- *Codegenerator* es una matriz k x n, donde cada elemento representa el valor octal correspondiente al polinomio generador o diagrama del codificador.

- *FeedbackConnection* es la variable que nos indica que es un codificador sistemático, y su valor es el del primer elemento del vector *codegenerator*.

El siguiente ejemplo corresponde al codificador sistemático convolucional utilizado para la creación de turbocódigos en este proyecto.

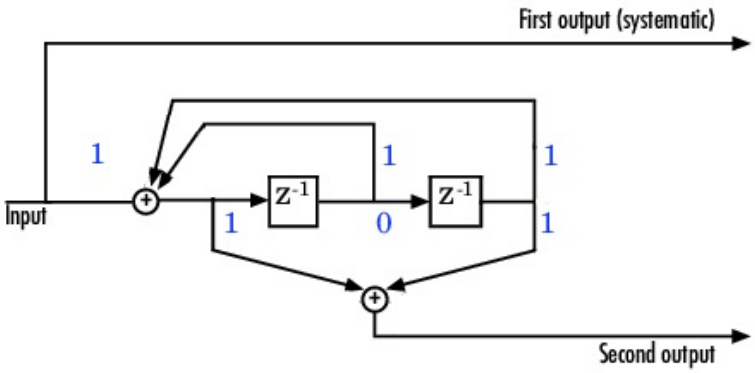


Figura 4.9: *Codificador convolucional sistemático*

Como podemos observar en la figura 4.9, nuestro codificador tiene una entrada y dos salidas. Una de ellas se corresponde con el valor de entrada, por lo que representa la salida sistemática. Así, al tener una sola entrada sólo tendremos una *constraint length* por de valor igual al número de registros más uno, o sea 3. En lo que a polinomio generador se refiere, nos hemos de fijar en las conexiones del diagrama del codificador representados por unos y 0 de color azul. Así, tenemos un polinomio generador de $g_0 = 111$ y $g_1 = 101$. Estos valores binarios se corresponden a octal en 7 y 5 respectivamente, por lo que el valor de la variable *codegenerator* es [7 5]. De aquí también obtenemos el valor de la variable *FeedbackConnection*, en nuestro caso será 7.

Una vez tenemos todos los valores definidos, invocamos la función poly2trellis para obtener el trellis asociado a nuestro codificador.

$$trellis = poly2trellis(3, [75], 7); \tag{4.25}$$

Como hemos dicho, la función trellis es de tipo struct y contiene los siguientes campos:

```
trellis=
        numInputSymbols: 2
        numOutputSymbols: 4
        numStates: 4
        nextStates: [4x2 double]
        outputs: [4x2 double]
```

En matlab se accede a cada uno de los campos de una variable struct mediante el punto. Así, en este caso observamos 4 valores dentro del trellis: *trellis.numInputSymbols, trellis.numOutputSymbols, trellis.numStates, trellis.nextStates y trellis.outputs.*

- *Trellis.numInputSymbols*. El número de símbolos de entrada. 2 indica el 0 y el 1, de modo que la entrada de símbolos en la secuencia recibida se contabiliza como $\log_2(trellis.numInputSymbols)$.

- *Trellis.numOutputSymbols*. El número de símbolos de salida, 4 indica 00, 01, 10 y 11. de modo que la salida de símbolos en la secuencia codificada se contabiliza como $\log_2(trellis.numOutputSymbols)$.

- *Trellis.numStates*. El número de estados, que son 4.

- *Trellis.nextstates*. Indica el estado siguiente según si la entrada es 0 o 1, y es una matriz donde la primera columna representa el estado siguiente si la entrada es un 0 y la segunda columna representa el estado siguiente si la entrada es un 0. El estado actual es el número de la fila. Los valores están en decimal.

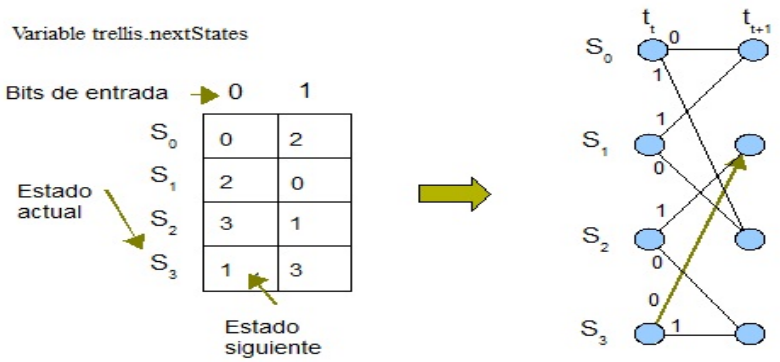


Figura 4.10: *Representación del trellis en matlab*

- *Trellis.outputs*. Indica la salida codificada para las transiciones de estado dadas por la variable trellis.nextStates. Su valor es en decimal, por ello se ha de pasar a binario para obtener la salida en bits.

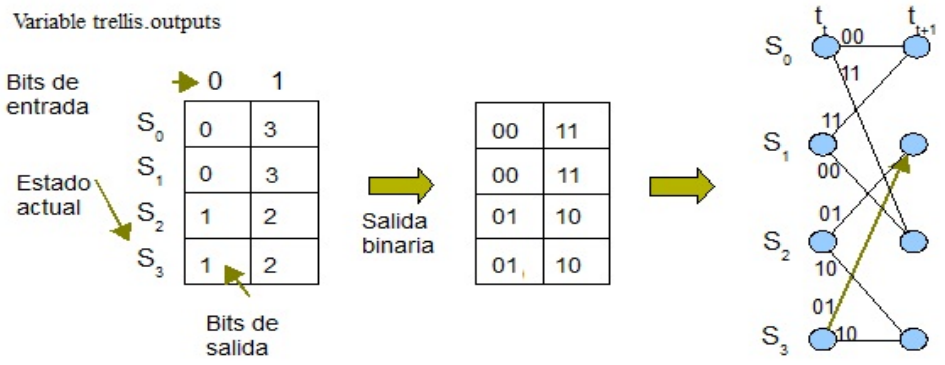


Figura 4.11: *Codificación del trellis en matlab*

Para la decodificación de códigos convolucionales en matlab existe una la función vitdec. Puesto que estamos en el tema de códigos convolucionales, mostraremos el funcionamiento de esta función, pero cabe puntualizar que no se ha utilizado en este proyecto, ya que en él los códigos convolucionales forman parte de la estructura de los códigos turbo, y es el proceso de turbocodificación el encargado de decodificar las secuencias de entrada codificadas.

- Función vitdec

Esta función decodifica la entrada utilizando el algoritmo de viterbi.

decoded= vitdec(***code, trellis, tblen, opmode, dectype***)

- ***Code*** es la secuencia de entrada,
- ***trellis*** es el trellis utilizado para la secuencia del código convolucional
- ***tblen*** es el retraso que transcurre antes de la primera decodificación.
- ***Opmode*** hay aquí tres posibilidades: *'cont'*, *'term'* y *'trunc'*
  - *'cont'*. El decodificador asume que ha empezado en el estado todo ceros, y, desde el punto con mejor métrica, vuelve del camino de trellis hacia atrás. El decodificador saca la primera secuencia decodificada una vez han transcurrido tblen instantes de tiempo.

- *'term'*. El decodificador asume que empieza y termina en el estado todo ceros. El decodificador parte del estado cero y recorre el camino del trellis hacia atrás. No hay retraso en la salida.
- *'trunc'*. El decodificador no asume que termina en el estado todo ceros, y tampoco hay una decodificación continua. Tampoco hay retraso tampoco en la salida de la secuencia decodificada.

- ***Dectype*** es el tipo de decodificación que queremos aplicar,
  - *'unquant'* la entrada consta de valores reales donde el -1 se corresponde con el 0 binario y 1 se corresponde con el 1 binario. 'hard' entrada binaria de 0 y 1.
  - 'soft' entrada real con valores reales.

Capítulo 5

Códigos Turbo

Berrou, Grlavieux y Thitimajshima dieron a conocer por primera vez este tipo de códigos en 1993 en el transcurso de la " IEEE International Conference on Communications" realizada en Ginebra, Suiza. Desde entonces, su uso se ha disparado, entre otras cosas por ser unos códigos que, bajo ciertas características se aproximan al máximo a la cota de Shannon, introducida en el tema 2. Esto es, según la teoría Shannon, la cantidad de información que puede ser conducida a través del canal por unidad de tiempo es limitada. Por otro lado, si queremos ofrecer robusteza a errores tendremos que añadir bits de redundancia en detrimento de bits de información. Los códigos turbo han demostrado ser los códigos que mayor capacidad de corrección tienen en relación a los bits de redundancia (o paridad) utilizados respecto a otro tipo de codificaciones. Esto es debido, básicamente, a dos factores clave de los códigos turbo: la primera es que su diseño permite crear códigos con propiedades pseudo-aleatorias, y esto, según la teoría de la información de Shannon es lo que permite sacar más provecho a la capacidad de canal; la segunda es el uso de la decisión indecisa a la salida soft-output soft-output en los algoritmos de decodificación iterativa.

Actualmente, los códigos turbo son ampliamente utilizados para la transmisión de datos codificados, ya que representan casi siempre una mejora en el aprovechamiento de la capacidad de canal. De este modo, son ampliamente utilizados en las comunicaciones espaciales, en la telefonía 3G, en la televisión digital etc.

5.1. Codificador turbo

Un codificador turbo, en su forma más simple, se constituye de dos codificadores convolucionales sistemáticos RSC en paralelo. Así, la entrada cada vez se codifica dos veces. La entrada al segundo codificador, a diferencia del primero va precedida de un reordenamiento de los bits mediante un etrelazador o interleaver. Este entrelazador juega un papel muy importante ya que es el que permite que las salidas sean estadísticamente independiente. Finalmente el codificador turbo consta de una salida sistemática, por lo que el ratio de codificación es $R_c = 1/3$. Veamos un esquema del codificador turbo en la siguiente figura:

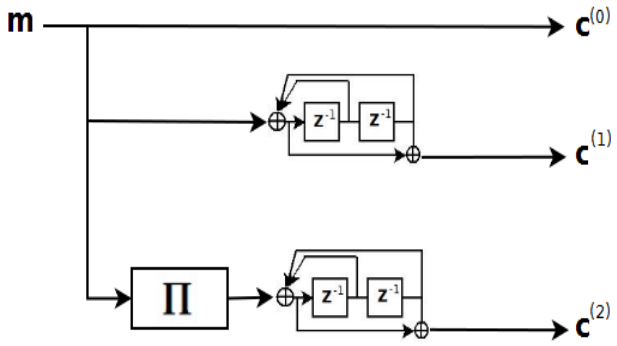


Figura 5.1: *Turbocodificador de ratio* $R = 1/3$

Los siguientes aspectos deben ser considerados en el diseño de un Turbo codificador:

- Pese a ser los codificadores convolucionales continuos, el proceso de decodificación, más aún si se utiliza el algoritmo MAP, conlleva un retardo considerable debido al precio computacional que conllevan los algoritmos de decodificación iterativa SISO. Así, los mensajes irán agrupados en tramas teniendo en cuenta que se presenta mejor rendimiento en los bloques de información de gran tamaño, usualmente de miles de bits.
- Para BER's del orden de 10^{-5}, el mejor rendimiento se logra mediante codificadores con constraint length bajas, normalmente menores de 4.
- Los codificadores convolucionales recursivos ofrecen mejores resultados que los no recursivos.

- Si se quiere reducir el ratio del código se puede utilizar la perforación de bits. Si se eliminan bits de paridad, por ejemplo cogiendo alternativamente $c^{(1)}$ y $c^{(2)}$, ofrece mejores resultados que eliminar bits sistemáticos, aunque también se pueden perforar bits de información.
- Se pueden crear códigos turbo múltiples, mediante la inserción de más codificadores RSC.
- Es preciso utilizar interleavers que ofrezcan un nivel de aleatoriedad mayor al mensaje.
- Como se ha comentado en el primer punto, la información se transmite por bloques debido al algoritmo de decodificación. En el caso del estudio de este proyecto, el algoritmo BJCR. Por ello, se generan señales que empiecen y terminen en el estado 0 del trellis.

Actualmente existen múltiples diseños de codificadores turbo según se distribuyan los codificadores convolucionales. Así, se pueden encontrar turbo códigos SCCC (Serial concatened convolutional code), PCCC (Paralel concatened convolutional code), PCCC múltiples e híbridos entre los dos HCCC (Hibrid concatened convolutional code). En la figura 5.2 se puede observar la geometría de estos cuatro tipos de codificadores turbo.

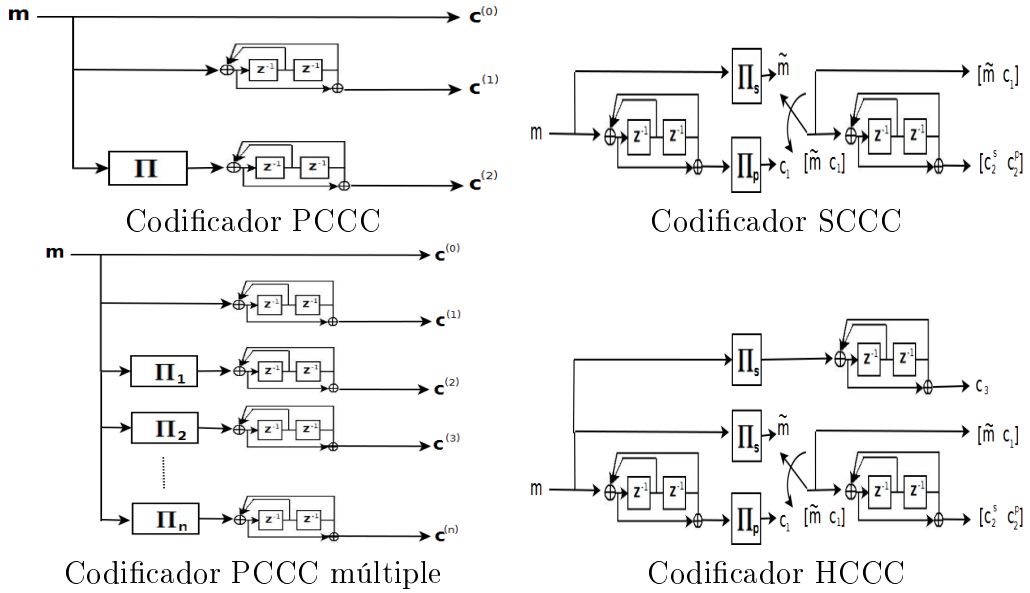


Figura 5.2: *Tipos de codificadores turbo según la distribución de los codificadores convolucionales*

Los siguientes gráficos muestran el comportamiento de los códigos SCCC y los códigos PCCC según la relación señal a ruido.

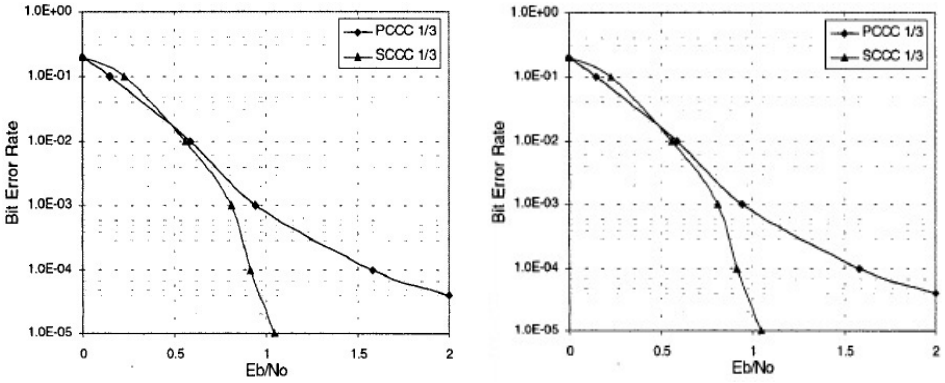


Figura 5.3: *a) Comparativa del comportamiento de los codificadores PCCC y SCCC de ratio 1/3, RSC de 4 estados, interleavers de 16384 bits y 9 iteraciones. b) comparativa del comportamiento de los codificadores PCCC y SCCC de ratio 1/3, RSC de 4 estados, interleavers de 1024 bits y 7 iteraciones*

La gráfica 5.3.a muestra como, para valores bajos de E_b/N_0, la distribucón paralela muestra mejores resultados, mientras que al ir aumentando la E_b/N_0 o la BER deseada, el codificador SCCC mejora el comportamiento del PCCC. Para valores de BER menores a 10^{-5}, el codificador SCCC mejora mucho el resultado y, además, no presenta el comportamiento asintótico del PCCC. En la figura b) se ha reducido el tamaño del interleaver y eso ha provocado que la asíntota del PCCC empezara a BERs mayores, y que el comportamiento del PCCC tan sólo sea mejor para E_b/N_0 menores de 0,5. Estos estudios están sacados del paper "Turbo code performance and design trade-offs" de Raffii Achiba y Mehrnaz Mortazavi. En concreto, las gráficas pertenecen a los trabajos de S. Benedetto, D.Divsalar G. Montorsi y F. Pollara.

Pese a que las comparativas y el diseño de los codificadores del estudio de Achiva y Mortazavi se han hecho para canales AWGN y Rayleigh, canales que no tienen el mismo comportamiento que el que en este estudio se pretende analizar, nos ponemos en el caso que tendremos BERs elevadas y SNR muy bajas y que, en tal caso, el codificador PCCC nos situará en una mejor situación de análisis, tal y como pasa con los canales AWGN y Rayleigh.

5.2. Decodificador turbo

El decodificador turbo, a diferencia del codificador, tiene una estructura compleja que utiliza un proceso iterativo por lo cual el cálculo final se simplifica y se hace posible. Más adelante profundizaremos en el algoritmo de decodificación MAP; por el momento diremos que cada decodificador utiliza los bits de entrada de canal juntamente una estimación de sobre la decodificación hecha por el otro decodificador, información *a priori*, esta información *a priori* es independiente de las otras entradas. Con estos datos el decodificador crea una secuencia de salida y una estimación que pasa al otro decodificador o LLR (*Log-Likelihood Ratio*). El siguiente decodificador utilizará esta información como información *a priori*. Es gracias a este proceso de retroalimentación que este tipo de códigos se denominan Códigos Turbo. A cada iteración, el decodificador se va acercando más al código transmitido habiendo un punto en el que por más iteraciones que realicemos nuestro decodificación no mejora. Hay que anotar que la información que se transmiten los decodificadores es soft (suave). La siguiente figura muestra el esquema de un decodificador turbo PCCC:

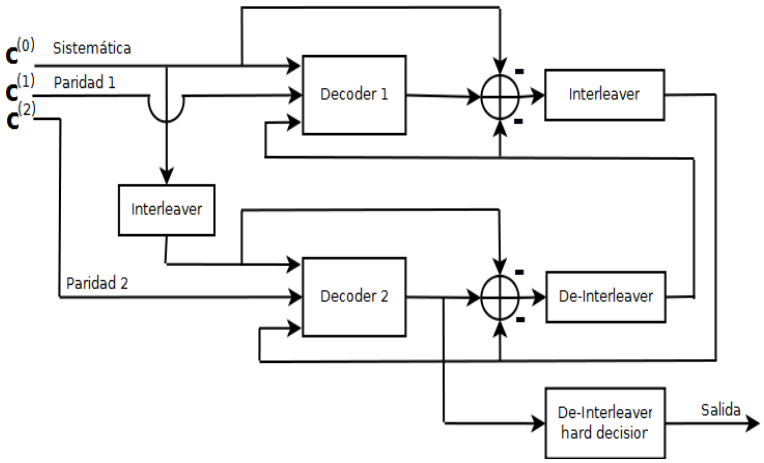


Figura 5.4: *Esquema de un decodificador turbo*

Como se muestra en la figura 5.4, en cada decodificador entran tres entradas, las procedentes del canal sistemática y de paridad, más la información procedente del otro decodificador. Con estos valores, mediante el algoritmo de decodificación (BJCR en nuestro caso), se genera una estimación sobre el código transmitido en el primer decodificador. Posteriormente, a esta in-

formación se le restan los valores sistemáticos y los valores aportados por el segundo decodificador para generar una salida "limpia" de estimación de los bits, que será transmitida al segundo codificador y que, a su vez, utilizará para realizar su predicción que transmitirá se al primer decodificador de nuevo. Resumiendo se podría decir que cada codificador hace lo mejor que puede con las entradas que tiene, y pasa su salida al otro decodificador para que, a su vez, haga también lo mejor que pueda. Presumiblemente en cada decodificación llevada a cabo por los decodificadores nos acercamos más hacia el resultado correcto llegando a un punto en que, o se alcanza el resultado correcto o los codificadores ya no pueden hacer nada más con la información que tienen y se da una decodificación errónea.

5.2.1. Algoritmo BCJR

En 1974 Bahl, Cocke, Jelinek y Raviv introdujeron un algoritmo de decodificación MAP llamado Algoritmo BCJR por sus iniciales. Este algoritmo, a diferencia del algoritmo de Viterbi, visto en la sección 4.4.1, nos permite obtener una descodificación MAP (máximum a posteriori) a nivel de bit recibido, mientras que el algoritmo de Viterbi nos proporcionaba la palabra código más probable de haber sido transmitida a partir de la distancia final del camino resultante en el Trellis, esto es se minimizaba la WER (word error rate). Se puede demostrar que optimizando la secuencia a nivel de bit (BER), también se optimiza la WER, no siendo así en el caso contrario. Por ello, el algoritmo BCJR tiene una rendimiento óptimo. Primero de todo, presentaremos una medida necesaria para explicar el algoritmo, LLR (likelyhood ratio), que se vendría a ser una medida de fiabilidad de una variable binaria aleatoria. Para más comodidad, la denominaremos este valor L, así:

$$L(b_i/\mathbf{r}) = \ln(\frac{P(b_i = +1/\mathbf{r})}{P(b_i = -1/\mathbf{r})}) \tag{5.1}$$

Esta ecuación se puede entender como la probabilidad de que el bit transmitido sea +1 o -1, dependiendo de la secuencia recibida a través del canal. Si la probabilidad de haber recibido +1 o -1 es similar, el valor de L será cercano al cero, haciéndose más negativo o positivo si las probabilidades de -1 o +1, respectivamente, se hacen predominantes. Esta estimación *a posteriori* de los bits b_i se basa en la estimación proporcionada por cada decodificador del decodificador turbo a modo de decisiones soft.

Trabajando nivel de bit podemos definir un LLR basado en la probabilidad de que la decodificación óptima del decodificador sea r_i si el bit transmitido es x_i:

$$L(r_i/x_i) = \ln(\frac{P(r_i/x_i = +1)}{P(r_i/x_i = -1)}) \tag{5.2}$$

Para el caso de un canal AWGN y de una secuencia de bits transmitidos en forma polar (+1,-1), definiremos la constante de canal del siguiente modo:

$$L(r_i/x_i) = \ln(\frac{e^{\frac{-E_b}{2\sigma^2}(r_i-1)^2}}{e^{\frac{-E_b}{2\sigma^2}(r_i+1)^2}}) = \frac{-E_b}{2\sigma^2}(r_i-1)^2 + \frac{-E_b}{2\sigma^2}(r_i+1)^2 = 2\frac{E_b}{2\sigma^2}r_i = L_c r_i \tag{5.3}$$

Esta constante de canal $L_c = \frac{E_b}{2\sigma^2}$ es una medida de la relación señal a ruido del canal y sirve para conocer el grado de fiabilidad que tienen las secuencias que recibimos de él.

Teniendo en cuenta que la probabilidad que un bit dado sea +1 o -1 depende de la suma de las probabilidades de transición entre estados del trellis que nos reporten el bit -1 o -1, se puede expresar la ecuación 5.2 como:

$$L(b_i/\mathbf{r}) = \ln \frac{\sum_{\{s',s\}\Rightarrow b_i=+1} P(S_{i-1} = s', S_i = s, \mathbf{r_1^n})}{\sum_{\{s',s\}\Rightarrow b_i=-1} P(S_{i-1} = s', S_i = s, \mathbf{r_1^n})} \tag{5.4}$$

Para obtener la ecuación anterior, se ha tenido que aplicar también el teorema de Bayes sobre la ecuación 5.2, donde $\mathbf{r}_1^n$ es la secuencia recibida hasta el instante n. Así, realizando las siguientes definiciones:

$$\begin{aligned} &\alpha_{i-1}(s') = P(S_{i-1} = s', \mathbf{r}_1^{i-1}) \\ &\gamma_i(s', s) = P(\{S_i = s, \mathbf{r}_i/S_{i-1} = s') \\ &\beta_i(s) = P(\mathbf{r}_{i+1}^n/S_i = s) \end{aligned} \tag{5.5}$$

Definimos también $\sigma_i(s', s) = P(S_{i-1} = s', S_i = s, \mathbf{r}_1^n)$; de esta forma:

$$\sigma_i(s', s) = \alpha_{i-1}(s')\gamma_i(s', s)\beta_i(s) \tag{5.6}$$

Así, la ecuación 5.4 nos queda de la siguiente manera:

$$L(b_i/\mathbf{r}) = \ln \frac{\sum_{\{s',s\}\Rightarrow b_i=+1} \alpha_{i-1}(s')\gamma_i(s', s)\beta_i(s)}{\sum_{\{s',s\}\Rightarrow b_i=-1} \alpha_{i-1}(s')\gamma_i(s', s)\beta_i(s)} \tag{5.7}$$

Llegados a este punto, podemos reescribir $\alpha_{i-1}(s')$ y $\beta_i(s)$ en función de $\gamma_i(s', s)$:

$$\alpha_{i-1}(s') = \sum_{s\in\sigma_{i-1}} \gamma_i(s', s)\alpha_{i-1}(s') \tag{5.8}$$

Estas $\alpha_{i-1}(s')$ se corresponden con las métricas hacia delante (*forward metric*); éstas se calculan en cada estado s en el instante $i-1$. Por ello, la expresión σ_{i-1} se refiere al conjunto de todos los estados en el instante i-1. Del mismo modo, utilizamos un cálculo en sentido contrario en el trellis para obtener la métrica hacia atrás (*backward metric*) $\beta_i(s)$:

$$\beta_i(s') = \sum_{s\in\sigma_i} \gamma_i(s', s)\beta_i(s') \tag{5.9}$$

Para poder realizar el cálculo recursivo de α y β, debemos definir unas condiciones iniciales, es decir, para las iteraciones primera y última, respectivamente:

$$\alpha_0(s) = \begin{cases} 1, & para\ s = 0 \\ 0, & para\ s \neq 0 \end{cases} \tag{5.10}$$

Para el caso de la métrica hacia atrás:

$$\beta_k(s) = \begin{cases} 1, & para\ s = 0 \\ 0, & para\ s \neq 0 \end{cases} \tag{5.11}$$

Describamos ahora los coeficientes de rama $\gamma_i(s', s)$, que se entienden como la probabilidad de pasar del estado s' a s en el instante i, habiendo recibido la secuencia $\mathbf{r}_i$:

$$\gamma_i(s', s) = p(s, \mathbf{r}_i|s') = p(\mathbf{r}_i|\{s', s\})p(b_i) = p(b_i)p(\mathbf{r}_i|\mathbf{x}_i) \tag{5.12}$$

La probabilidad $p(b_i)$ se puede definir en función del LLR de b_i, $L(b_i)$:

$$P(b_i = \pm 1) = \frac{e^{-L(b_i)/2}}{1 + e^{-L(b_i)}} e^{L(b_i)/2} = A_i e^{L(b_i)/2} \tag{5.13}$$

Por otra parte,$p(\mathbf{r}_i|\mathbf{x}_i)$ se obtiene bajo la suposición de canal AWGN de la siguiente forma:

$$\begin{aligned} p(\mathbf{r}_i|\mathbf{x}_i) &= \prod_{k=1}^{n} P(r_{ik}/x_{ik}) = \prod_{k=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{E_b}{2\sigma^2}\sum_{k=1}^{n}(r_{ik}-x_{ik})^2} \\ &= \frac{1}{(\sqrt{2\pi})^n} e^{-\frac{E_b}{2\sigma^2}\sum_{k=1}^{n}(r_{ik}^2+x_{ik}^2)} e^{-\frac{E_b}{2\sigma^2}\sum_{k=1}^{n} r_{ik}x_{ik}} = B_i e^{\frac{E_b}{\sigma^2}\sum_{k=1}^{n} r_{ik}x_{ik}} \end{aligned} \tag{5.14}$$

Así, el coeficiente $\gamma_i(s', s)$ queda definido mediante la siguiente ecuación:

$$\gamma_i(s', s) = C e^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}, \ \textit{donde } C = A_i B_i \tag{5.15}$$

De este modo, en esta ecuación tenemos todos los parámetros necesarios para encontrar los valores de los coeficientes implicados en el algoritmo, pues recordemos que los coeficientes α_i y β_i se obtienen en función de γ_i. A parte de la secuencia recibida, hemos de destacar la información L_c, que es la constante de canal y nos vendría a definir, como hemos apuntado anteriormente, cuánto nos podemos fiar de los bits que llegan a través del canal. El otro parámetro $L(b_i)$ representa la información *a priori* para el decodificador en curso proporcionada como información extrínseca por el otro decodificador.

En este punto ya estamos a disposición de desarrollar la ecuación 5.7, es decir, el LLR del bit b_i. Después de recibir la secuencia $\mathbf{r}$, una vez hecho esto para toda la secuencia, es decir desde $i = 0$ hasta $i = n$, tendremos la estimación de la secuencia transmitida.

$$
\begin{aligned}
L(b_i/\mathbf{r}) &= \ln \frac{\sum_{\{s',s\}\Rightarrow b_i=+1} \alpha_{i-1}(s')\gamma_i(s',s)\beta_i(s)}{\sum_{\{s',s\}\Rightarrow b_i=-1} \alpha_{i-1}(s')\gamma_i(s',s)\beta_i(s)} \\
&= \ln \frac{\sum_{\{s',s\}\Rightarrow b_i=+1} \alpha_{i-1}(s')e^{L(b_i)/2}e^{L_c r_{i1}/2}\gamma_{iextr}(s',s)\beta_i(s)}{\sum_{\{s',s\}\Rightarrow b_i=-1} \alpha_{i-1}(s')e^{-L(b_i)/2}e^{-L_c r_{i1}/2}\gamma_{iextr}(s',s)\beta_i(s)} \\
&= L(b_i) + L_c r_{i1} + L_e(b_i)
\end{aligned}
\tag{5.16}
$$

Notar que, debido a que el cálculo de los LLR se realiza mediante el logaritmo de un cociente, los términos constantes se eliminan. Aclarar también que el término $\mathbf{r}_{i1}$ es debido a la naturaleza de los codificadores convolucionales, pues al tener más de una codificación para un bit dado (bit de paridad más bit de redundancia), es posible separarlos y, en este caso, mediante $\mathbf{r}_{i1}$, nos referimos al primer bit codificado por el RSC, el de paridad. Además, en la ecuación anterior podemos separar la información intrínseca con la información extrínseca, que es la que se transmite al siguiente decodificador:

$$
L_e(b_i) = \ln \frac{\sum_{\{s',s\}\Rightarrow b_i=+1} \alpha_{i-1}(s')\gamma_{iextr}(s',s)\beta_i(s)}{\sum_{\{s',s\}\Rightarrow b_i=-1} \alpha_{i-1}(s')\gamma_{iextr}(s',s)\beta_i(s)} \tag{5.17}
$$

La información transmitida de un decodificador a otro en cada iteración será por lo tanto:

$$
L_e(b_i) = L(b_i/\mathbf{r}) - L(b_i) - L_c r_{i1} \tag{5.18}
$$

Resumiendo, los pasos a seguir en el algoritmo BCJR en el dominio logarítmico (Log-domain BCJR) son:

1. Se inicializan las metricas α_0 y β_k.
2. Se calculan las métricas de rama $\gamma_i(s',s)$ para toda la secuencia $i = 0, 1, \ldots, k-1$.
3. Se calculan las métricas hacia delante $\alpha_{i-1}(s)$, para $i = 0, 1, \ldots, k-1$.
4. Se calculan las métricas hacia atrás $\beta_i(s')$, para $i = k-1, k-2 \ldots, 0$.

5. Se obtienen los valores LLR, ecuación 5.16.

6. Se realizan las decisiones hard.

5.2.2. Proceso iterativo de decodificación turbo mediante el algoritmo Log-Map

Recuperando la figura 5.4, en la siguiente figura podemos plasmar el paso de valores LLR utilizados en el algoritmo log-Map BCJR en cada uno de los decodificadores que constituyen el decodificador turbo:

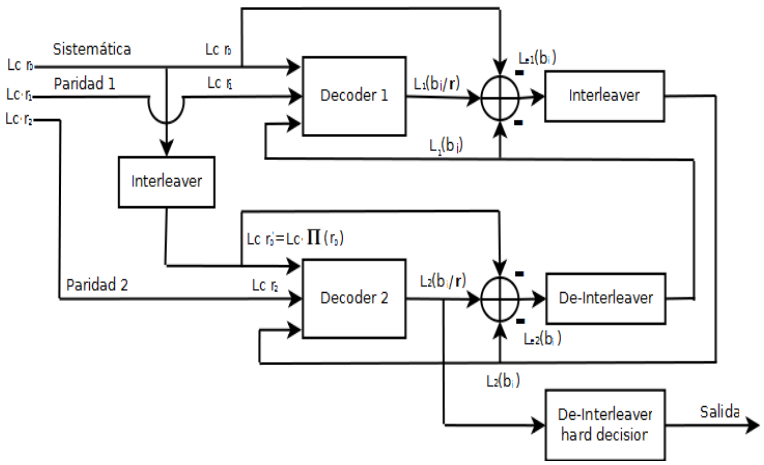


Figura 5.5: *Esquema de un decodificador turbo*

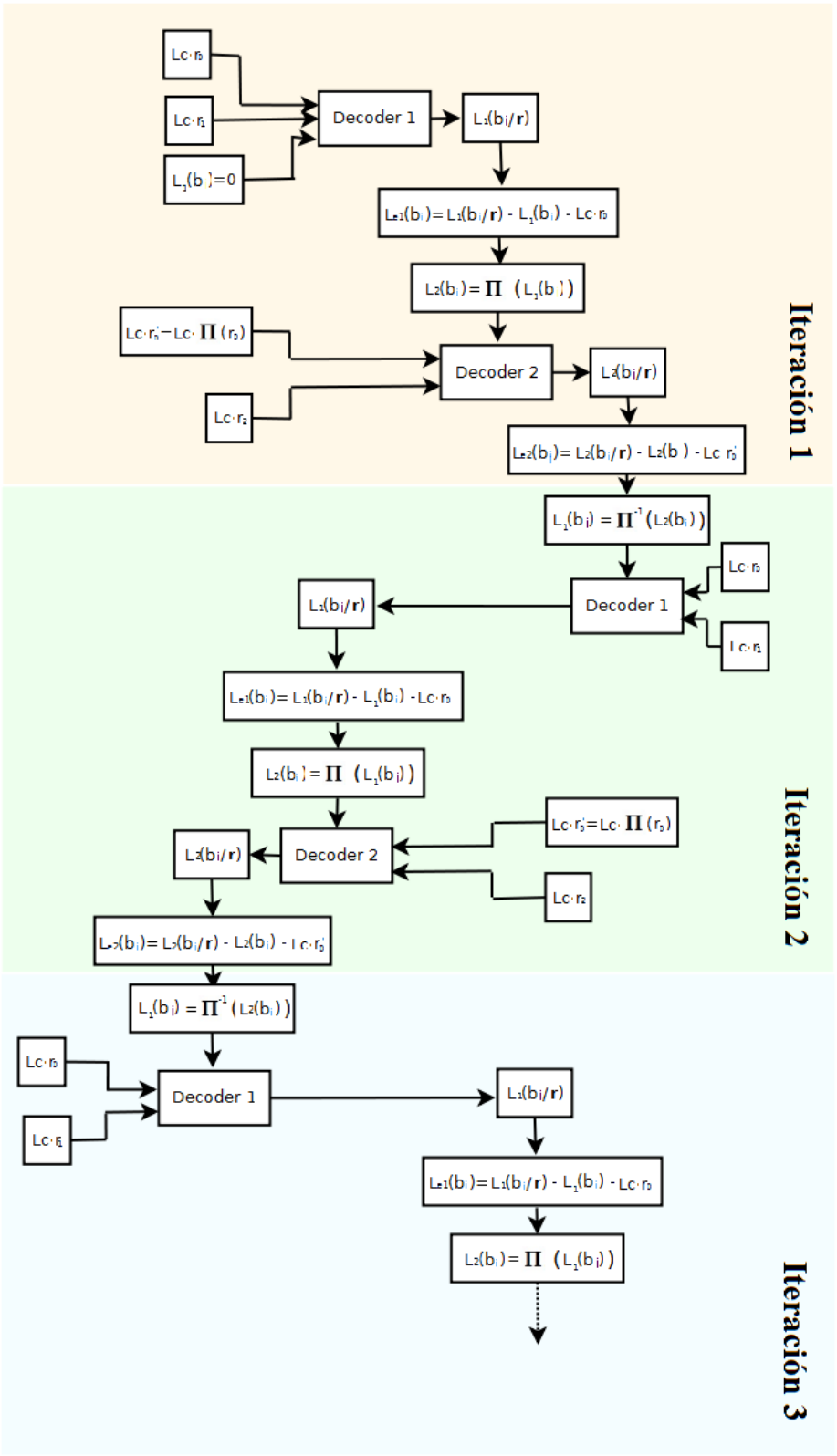


Figura 5.6: *desglose lineal del proceso de decodificación iterativo en el decodificador turbo*

5.2.3. Ejemplo de una codificación turbo

En el siguiente ejemplo realizaremos una codificación turbo de un mensaje de 12 bits, le añadiremos ruido gausiano y, posteriormente, lo decodificaremos mediante el algoritmo log-BCJR. Todo el proceso de codificación y decodificación de la secuencia utilizada se ha realizado mediante el codificador turbo diseñado en Matlab para este proyecto. Éste será comentado con más detalle en el próximo capitulo.

Ejemplo 7 *Ejemplo práctico de la decodificación turbo mediante el algoritmo log-BCJR. El trellis utilizado es el de la figura 4.10 y 4.11.*

$Mensaje: msg = (1,1,0,1,1,0,0,1,1,1,0,1)$

$interleaver = (10,4,5,2,9,7,3,11,6,12,1,8)$

deinterleaver =(11,4,7,2,3,9,6,12,5,1,8,10)

Los pasos a seguir en este ejemplo serán los siguientes:

1. *Generamos la secuencia codificada de ratio* $\frac{1}{3}$.
2. *Añadimos ruido a la secuencia codificada.*
3. *Decodificamos la secuencia parándonos en cada codificador y en cada iteracion para poder sacar los valores del algoritmo.*

Para codificar el mensaje utilizamos la función turbocod:

$$\mathbf{x} = \mathbf{turbocod}(\mathbf{msg}, \mathbf{trellis});$$

Una vez añadidos los bits de cola para que el mensaje termine y empiece en el estado 0 los trellis asociados a cada codificador RSC del codificador turbo son los siguientes:

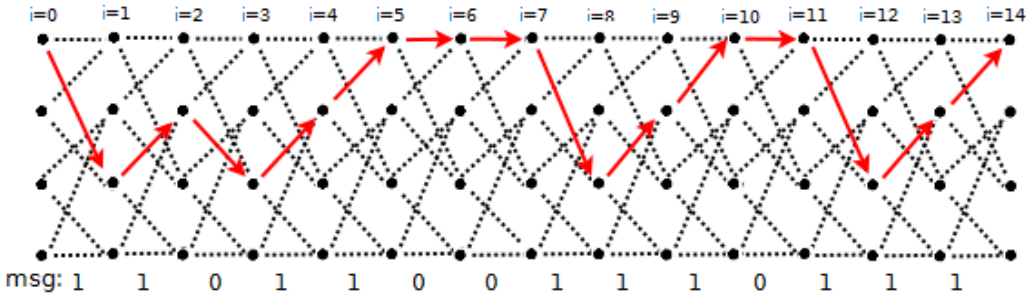


Figura 5.7: *Trellis correspondiente a la codificación convolucional RSC del codificador 1 sobre el mensaje una vez añadidos los bits de cola.*

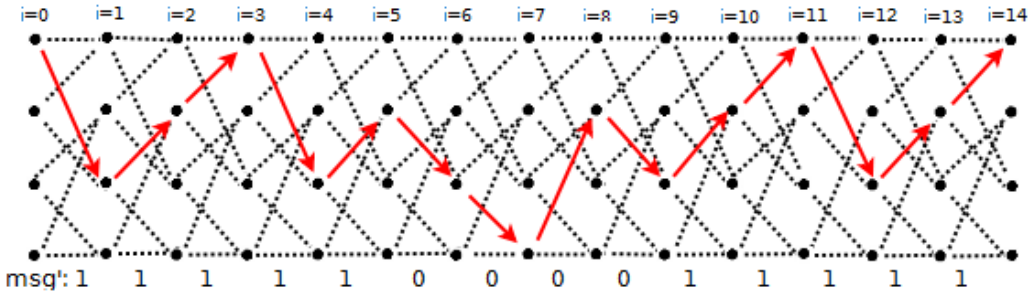


Figura 5.8: *Trellis correspondiente a la codificación convolucional RSC del codificador 2 sobre el mensaje una vez entrelazado el mensaje de entrada y añadidos los bits de cola.*

Finalmente obtenemos los bits codificados por los dos codificadores RSC, colocados en forma bipolar y entrelazados de nuevo mediante un entrelazador matricial:

x = (1,1,1,1,-1,-1,-1,-1,1,1, -1, 1, 1, 1, -1, -1, -1, -1, -1, -1, 1, 1, 1, 1, 1, -1, -1, 1, 1, -1, -1, -1, 1, 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, -1, -1, 1, -1, -1)

Aunque la tasa de codificación es de $\frac{1}{3}$, aquí se muestran 48 bits debido a los bits de cola añadidos para que el trellis empiece y termine en el estado 0 en cada uno de los codificadores.

Para añadir ruido a la secuencia utilizaremos la función de matlab ***awgn(x,SNR)****, donde el primer argumento es la secuencia de entrada y el segundo argumento es el nivel de SNR deseada. Para este ejemplo hemos utilizado una potencia de señal a ruido de -2dB.*

$$\mathbf{r} = \mathbf{awgn}(\mathbf{x}, -\mathbf{2});$$

r =(0.6208,0.1204,2.0484,0.1255,-1.5815,0.1124,-0.4512, 0.1289,1.6354,0.4953, -1.6469,2.0026,0.1550,2.4939,-0.0046,-0.6378,-0.9959,-0.5397,3.4398,-1.1415, -0.9596,3.4110,1.7678,0.1843,4.2950,-0.3064,-0.6296,0.0208,-0.3407,-3.2263,-1.5324, -2.3258,1.8155,0.6001,3.2270,2.9017,1.2065, -1.3560,0.4505, -0.4434, 1.8481,0.1576,0.4960, -1.8457,-0.2753,0.0204,-2.3389,-0.3038).

Una vez llegada la secuencia del canal, queda desglosada en bits de paridad y sistemáticos de la siguiente forma, se debe tener en cuenta que hay bits de paridad incluidos en el proceso de codificación y que se deben tratar en el decodificador en recepción, ya que no incluyen información aplicable al algoritmo MAP.

$r_0(sistematico)$ *=(0.6208, 0.1255, -0.4512, 0.4953, 0.1550, -0.6378, 3.4398, 3.4110, 4.2950, 0.0208, -1.5324, 0.6001, 1.2065, -0.4434, 0.4960, 0.0204)*

$r_1(paridad1)$ *=(0.1204, -1.5815, 0.1289, -1.6469, 2.4939, -0.9959, -1.1415, 1.7678, -0.3064, -0.3407, -2.3258, 3.2270, -1.3560, 1.8481, -1.8457, -2.3389)*

$r_2(paridad2)$ *=(2.0484, 0.1124, 1.6354, 2.0026, -0.0046, -0.5397, -0.9596, 0.1843, -0.6296, -3.2263, 1.8155, 2.9017, 0.4505, 0.1576, -0.2753, -0.3038)*

Como último paso invocaremos la función diseñada ***decturbo****, la cual realizará los cálculos de 5.16-5.18 e irá creando las informaciones apriori y extrínseca para después transmitirlas en cada iteración de un decodificador al otro. Como hemos dicho, nos detendremos en cada uno de los decoders para ver los valores que van tomando las variables γ, α y beta más una tercera variable σ que se corresponde al numerador y denominador, dependiendo si se hace el cálculo de los unos y zeros, de la ecuación 5.7.*

$$[\mathbf{x_{soft}}\ \mathbf{x_{hard}}] = \mathbf{decturbo}(\mathbf{r}, \mathbf{1}, \mathbf{trellis}, \mathbf{interleaver}, \mathbf{4})$$

donde el segundo argumento se corresponde a la constante de canal $L_c = 1$.

Iteración 1, decodificador 1

Cálculo de la γ para los valores de entrada 0 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}$														
$\gamma(0,0)$	0.69	2.07	1.17	1.77	0.26	2.26	0.31	0.07	0.13	1.17	6.88	0.14	1.07	0.49
$\gamma(1,2)$	0.69	2.07	1.17	1.77	0.26	2.26	0.31	0.07	0.13	1.17	6.88	0.14	1.07	0.49
$\gamma(2,3)$	0.77	0.42	1.33	0.34	3.22	0.83	0.10	0.43	0.10	0.83	0.67	3.71	0.27	3.14
$\gamma(3,1)$	0.77	0.42	1.33	0.34	3.22	0.83	0.10	0.43	0.10	0.83	0.67	3.71	0.27	3.14
Cálculo de la γ para los valores de entrada 1 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}$														
$\gamma(0,2)$	1.44	0.48	0.85	0.56	3.76	0.44	3.15	13.32	7.34	0.85	0.14	6.77	0.92	2.01
$\gamma(2,0)$	1.44	0.48	0.85	0.56	3.76	0.44	3.15	13.32	7.34	0.85	0.14	6.77	0.92	2.01
$\gamma(3,1)$	1.28	2.34	0.74	2.91	0.31	1.19	9.88	2.27	9.98	1.19	1.48	0.26	3.60	0.31
$\gamma(1,3)$	1.28	2.34	0.74	2.91	0.31	1.19	9.88	2.27	9.98	1.19	1.48	0.26	3.60	0.31

Cuadro 5.1: *Valores γ para las distintas transiciones del trellis del decodificador 1 en la iteración 1*

Cálculo de la γ_e para los valores de entrada 0 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,0)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	0.41	1.16	1.18	3.19	0.19	1.96	0.39
$\gamma_e(1,2)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	0.41	1.16	1.18	3.19	0.19	1.96	0.39
$\gamma_e(2,3)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	2.42	0.85	0.84	0.31	5.02	0.50	2.51
$\gamma_e(3,1)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	2.42	0.85	0.84	0.31	5.02	0.50	2.51
Cálculo de la γ_e para los valores de entrada 1 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,2)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	2.42	0.85	0.84	0.31	5.02	0.50	2.51
$\gamma_e(2,0)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	2.42	0.85	0.84	0.31	5.02	0.50	2.51
$\gamma_e(3,1)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	0.41	1.16	1.18	3.19	0.19	1.96	0.39
$\gamma_e(1,3)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	0.41	1.16	1.18	3.19	0.19	1.96	0.39

Cuadro 5.2: *Valores γ_e para las distintas transiciones del trellis del decodificador 1 en la iteración 1*

Cálculo de $\alpha_{i-1}(s') = \frac{1}{N}\sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$															
$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\alpha(0)$	1	0.32	0.24	0.38	0.26	0.46	0.45	0.12	0.59	0.09	0.16	0.25	0.14	0.07	0.27
$\alpha(1)$	0		0.58	0.09	0.46	0.12	0.18	0.34	0.09	0.26	0.35	0.10	0.11	0.39	0.47
$\alpha(2)$	0	0.67	0.05	0.44	0.13	0.28	0.19	0.26	0.21	0.52	0.19	0.52	0.34	0.07	0.11
$\alpha(3)$	0		0.10	0.07	0.13	0.12	0.15	0.27	0.09	0.11	0.28	0.11	0.39	0.45	0.12

Cuadro 5.3: *Cálculo de las α normalizadas por el factor* $N = \sum_{s'\in\sigma_{i-1}} \sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$

Cálculo de $\beta_i(s') = \frac{1}{N}\sum_{s\in\sigma_i}\gamma_i(s',s)\beta_i(s')$														
$i=$	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\beta(0)$	0.37	0.31	0.43	0.41	0.16	0.23	0.66	0.02	0.30	2.08	0.61	0.03	0.14	1
$\beta(1)$	0.35	0.31	0.36	0.25	0.18	0.87	0.06	0.28	0.22	0.13	0.05	0.02	0.59	0
$\beta(2)$	0.40	0.29	0.40	0.28	0.29	0.27	0.11	0.28	0.02	0.05	0.02	0.42	0	0
$\beta(3)$	0.37	0.28	0.38	0.23	0.22	0.13	0.03	0.02	0.01	0.03	0.02	0.03	0	0

Cuadro 5.4: *Cálculo de las β normalizadas por el factor $N = \sum_{s'\in\sigma_i}\sum_{s\in\sigma_i}\gamma_i(s',s)\beta_i(s')$*

Cálculo de la σ para los valores de entrada 0 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	0.35	0.22	0.10	0.36	0.01	0.18	0.53	0.00	0.20	0.22	0.32	0.00	0.04	0.03
$\sigma(1)$	0	0	0.22	0.05	0.04	0.05	0.03	0.04	0.00	0.01	0.03	0.00	0	0
$\sigma(2)$	0	0.08	0.02	0.04	0.10	0.02	0.00	0.01	0.00	0.01	0.00	0.08	0	0
$\sigma(3)$	0	0	0.04	0.00	0.08	0.06	0.00	0.19	0.01	0.01	0.00	0.01	0.12	0
Cálculo de la σ para los valores de entrada 1 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	0.42	0.04	0.10	0.04	0.27	0.07	0.02	0.08	0.01	0.00	0.00	0.53	0	0
$\sigma(1)$	0	0	0.27	0.01	0.27	0.01	0.06	0.02	0.02	0.46	0.06	0.01	0.01	0.99
$\sigma(2)$	0	0.46	0.01	0.25	0.00	0.41	0.02	0.03	0.05	0.08	0.03	0.00	0.40	0
$\sigma(3)$	0	0	0.03	0.04	0.00	0.02	0.00	0.00	0.00	0.00	0.02	0.00	0	0

Cuadro 5.5: *Valores σ para las distintas transiciones del trellis del decodificador 1 en la iteración 1*

Una vez tenemos todos los valores, estamos en disposición de completar la ecuación 5.16 para encontrar los bits codificados por el codificador uno en la primera iteración:

$L(b_i/\mathbf{r}) = L(b_i)+L_c r_{i1}+L_e(b_i)$ *=(0.8124, 0.6122, -0.3274, 0.2128, 0.9761, -0.1409, 1.9443, 2.8581, 3.3997, 0.7428, -2.5751, 2.1917, 2.1351, 3.0367)*

Donde,

$$L(b_i) = \frac{\sigma_i(s)_1}{\sigma_i(s)_0} = \frac{\sum_{\{s',s\}\Rightarrow b_i=+1}\alpha_{i-1}(s')\gamma_{extr}(s',s)_1\beta_k(s)}{\sum_{\{s',s\}\Rightarrow b_i=-1}\alpha_{i-1}(s')\gamma_{extr}(s',s)_0\beta_k(s)}$$

Así el valor extrínseco que se transmite al otro decodificador $L_e(b_i)$ es:

L_{e1} *= (0.1916, 0.4867, 0.1238, -0.2825, 0.8211, 0.4969, -1.4955, -0.5529, -0.8953, 0.7221, -1.0426, 1.5916, 0, 0)*

Iteración 1, decodificador 2

Cálculo de la γ para los valores de entrada 0 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i\cdot\mathbf{x}_i}$														
$\gamma(0,0)$	0.24	0.84	0.27	0.27	0.18	0.49	1.90	3.30	1.47	1.67	0.26	0.05	0.62	0.91
$\gamma(1,2)$	0.24	0.84	0.27	0.27	0.18	0.49	1.90	3.30	1.47	1.67	0.26	0.05	0.62	0.91
$\gamma(2,3)$	1.92	0.95	1.39	2.00	0.18	0.28	0.72	3.97	0.78	0.06	1.65	1.02	0.97	1.07
$\gamma(3,1)$	1.92	0.95	1.39	2.00	0.18	0.28	0.72	3.97	0.78	0.06	1.65	1.02	0.97	1.07
Cálculo de la γ para los valores de entrada 1 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i\cdot\mathbf{x}_i}$														
$\gamma(0,2)$	4.03	1.17	3.69	3.69	5.46	2.01	0.52	0.30	0.68	0.59	3.72	17.81	1.60	1.09
$\gamma(2,0)$	4.03	1.17	3.69	3.69	5.46	2.01	0.52	0.30	0.68	0.59	3.72	17.81	1.60	1.09
$\gamma(3,1)$	0.52	1.05	0.71	0.49	5.48	3.46	1.37	0.25	1.27	15.01	0.60	0.97	1.02	0.93
$\gamma(1,3)$	0.52	1.05	0.71	0.49	5.48	3.46	1.37	0.25	1.27	15.01	0.60	0.97	1.02	0.93

Cuadro 5.6: *Valores γ para las distintas transiciones del trellis del decodificador 2 en la iteración 1*

Cálculo de la γ_e para los valores de entrada 0 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,0)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92
$\gamma_e(1,2)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92
$\gamma_e(2,3)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
$\gamma_e(3,1)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
Cálculo de la γ_e para los valores de entrada 1 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,2)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
$\gamma_e(2,0)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
$\gamma_e(3,1)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92
$\gamma_e(1,3)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92

Cuadro 5.7: *Valores γ_e para las distintas transiciones del trellis del decodificador 2 en la iteración 1*

Cálculo de $\alpha_{i-1}(s') = \frac{1}{N}\sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$															
$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\alpha(0)$	1	0.05	0.02	0.59	0.25	0.08	0.42	0.41	0.36	0.29	0.07	0.47	0.35	0.11	0.2226
$\alpha(1)$	0		0.49	0.21	0.07	0.59	0.30	0.12	0.13	0.21	0.38	0.27	0.01	0.30	0.2865
$\alpha(2)$	0	0.94	0.03	0.07	0.61	0.24	0.15	0.34	0.14	0.21	0.06	0.12	0.61	0.27	0.2057
$\alpha(3)$	0		0.44	0.12	0.05	0.07	0.11	0.11	0.35	0.26	0.47	0.12	0.01	0.29	0.2852

Cuadro 5.8: *Cálculo de las α normalizadas por el factor $N = \sum_{s'\in\sigma_{i-1}} \sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$*

Cálculo de $\beta_i(s') = \frac{1}{N}\sum_{s\in\sigma_i}\gamma_i(s',s)\beta_i(s')$														
$i=$	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\beta(0)$	0.10	0.33	.14	0.13	0.06	0.15	0.19	0.13	0.01	0.01	0.22	0.01	0.43	1
$\beta(1)$	0.11	0.28	.17	0.10	0.06	0.12	0.07	0.27	0.01	0.10	0.11	0.05	0.52	0
$\beta(2)$	0.10	0.13	.16	0.10	0.08	0.15	0.13	0.04	0.71	0.01	0.02	0.03	0	0
$\beta(3)$	0.10	0.15	.10	0.23	0.15	0.24	0.47	0.06	0.17	0.02	0.02	0.03	0	0

Cuadro 5.9: *cálculo de las β normalizadas por el factor $N = \sum_{s'\in\sigma_i}\sum_{s\in\sigma_i}\gamma_i(s',s)\beta_i(s')$*

Cálculo de la σ para los valores de entrada 0 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	0.03	0.01	0.00	.02	0.01	0.01	0.13	0.05	0.00	0.02	0.01	0.00	0.12	0.11
$\sigma(1)$	0		0.03	.00	0.00	0.12	0.06	0.00	0.13	0.01	0.01	0.00	0	0
$\sigma(2)$	0	0.15	0.00	.04	0.09	0.04	0.04	0.02	0.01	0.00	0.01	0.01	0	0
$\sigma(3)$	0		0.17	.03	0.00	0.00	0.00	0.03	0.00	0.00	0.13	0.02	0.01	0
Cálculo de la σ para los valores de entrada 1 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	0.29	0.01	0.01	0.17	0.02	0.01	0.03	0.01	0.19	0.00	0.01	0.07	0	0
$\sigma(1)$	0	0	0.15	0.07	0.00	0.06	0.03	0.01	0.00	0.00	0.21	0.02	0.01	0.33
$\sigma(2)$	0	0.25	0.00	0.00	0.04	0.04	0.01	0.08	0.00	0.10	0.00	0.00	0.25	0
$\sigma(3)$	0	0	0.02	0.01	0.00	0.02	0.08	0.00	0.08	0.03	0.01	0.00	0	0

Cuadro 5.10: *Valores σ para las distintas transiciones del trellis del decodificador 2 en la iteración 1*

Una vez tenemos todos los valores estamos en disposición de completar la ecuación 5.16 para encontrar los bits codificados por el segundo codificador en la primera iteración:

$L(b_i/\mathbf{r}) = L(b_i)+L_c r_{i1}+L_e(b_i)$ = *(2.8060, 0.6176, 0.8415, 1.4157, 2.9432, 1.6546, -0.6740, -2.4389, 0.3690, 3.2178, 1.2134, 3.5399, 1.1720, 1.1286)*

Donde,

$$L(b_i) = \frac{\sigma_i(s)_1}{\sigma_i(s)_0} = \frac{\alpha_{i-1}(s')\gamma_{extr}(s',s)_1\beta_k(s)}{\alpha_{i-1}(s')\gamma_{extr}(s',s)_0\beta_k(s)}$$

Así el valor extrínseco que se transmite al otro decodificador $L_e(b_i)$ es:

L_{e2} = *(2.0632, 0.4049, -0.1346, 0.8035, -0.4565, -0.2896, -0.3466, 0.1362, 0.5099, 1.0261, 0.4010, 0.6819, 2.1351, 3.0367)*

Si termináramos aquí la codificación, el valor a la salida del decodificador 2 sería el siguiente:

deinterleaver$(L(b_i/\mathbf{r}))$ *= (1.2134, 1.4157, -0.6740, 0.6176, 0.8415, 0.3690, 1.6546, 3.5399 2.9432, 2.8060, -2.4389, 3.2178)*

Por lo que el decisor devolvería la siguiente decodificación:

x' *=(1, 1, -1, 1, 1, 1, 1, 1, 1, 1, -1, 1)*

x *=(1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1)*

Así, las posiciones 6 y 7 de la decodificación son erróneas.

Iteración 2, decodificador 1

Cálculo de la γ para los valores de entrada 0 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}$														
$\gamma(0,0)$	0.56	1.38	1.39	1.45	0.28	1.75	0.36	0.05	0.17	0.41	6.43	0.08	0.37	0.10
$\gamma(1,2)$	0.56	1.38	1.39	1.45	0.28	1.75	0.36	0.05	0.17	0.41	6.43	0.08	0.37	0.10
$\gamma(2,3)$	0.63	0.28	1.58	0.27	3.44	0.64	0.11	0.31	0.12	0.29	0.62	2.22	0.09	0.68
$\gamma(3,1)$	0.63	0.28	1.58	0.27	3.44	0.64	0.11	0.31	0.12	0.29	0.62	2.22	0.09	0.68
Cálculo de la γ para los valores de entrada 1 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}$														
$\gamma(0,2)$	1.77	0.72	0.71	0.68	3.51	0.57	2.73	18.73	5.84	2.39	0.15	11.32	2.69	9.21
$\gamma(2,0)$	1.77	0.72	0.71	0.68	3.51	0.57	2.73	18.73	5.84	2.39	0.15	11.32	2.69	9.21
$\gamma(3,1)$	1.56	3.50	0.62	3.57	0.29	1.54	8.54	3.19	7.94	3.36	1.59	0.44	10.47	1.45
$\gamma(1,3)$	1.56	3.50	0.62	3.57	0.29	1.54	8.54	3.19	7.94	3.36	1.59	0.44	10.47	1.45

Cuadro 5.11: *Valores γ para las distintas transiciones del trellis del decodificador 1 en la iteración 2*

Cálculo de la γ_e para los valores de entrada 0 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,0)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	0.41	1.16	1.18	3.19	0.19	1.96	0.39
$\gamma_e(1,2)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	0.41	1.16	1.18	3.19	0.19	1.96	0.39
$\gamma_e(2,3)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	2.42	0.85	0.84	0.31	5.02	0.50	2.51
$\gamma_e(3,1)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	2.42	0.85	0.84	0.31	5.02	0.50	2.51
Cálculo de la γ_e para los valores de entrada 1 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,2)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	.42	0.85	0.84	0.31	5.02	0.50	2.51
$\gamma_e(2,0)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	.42	0.85	0.84	0.31	5.02	0.50	2.51
$\gamma_e(3,1)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	.41	1.16	1.18	3.19	0.19	1.96	0.39
$\gamma_e(1,3)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	.41	1.16	1.18	3.19	0.19	1.96	0.39

Cuadro 5.12: *Valores γ_e para las distintas transiciones del trellis del decodificador 1 en la iteración 2*

Cálculo de $\alpha_{i-1}(s') = \frac{1}{N}\sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$															
$i=$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\alpha(0)$	1	0.24	0.09	0.32	0.16	0.59	0.48	0.12	0.64	0.08	0.18	0.23	0.09	0.02	0.82
$\alpha(1)$	0	0	0.78	0.06	0.62	0.11	0.16	0.39	0.08	0.24	0.58	0.04	0.08	0.59	0.04
$\alpha(2)$	0	0.75	0.05	0.54	0.10	0.19	0.23	0.27	0.20	0.57	0.09	0.67	0.51	0.03	0.04
$\alpha(3)$	0	0	0.06	0.05	0.11	0.09	0.12	0.21	0.06	0.08	0.13	0.04	0.30	0.34	0.08

Cuadro 5.13: *Cálculo de las α normalizadas por el factor* $N = \sum_{s'\in\sigma_{i-1}} \sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$

Cálculo de $\beta_i(s') = \frac{1}{N}\sum_{s\in\sigma_i} \gamma_i(s',s)\beta_i(s')$														
$i=$	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\beta(0)$	1.48	1.86	3.42	4.15	0.79	1.40	7.49	0.06	0.51	8.00	4.20	0.00	0.01	1
$\beta(1)$	1.30	1.60	2.57	1.20	0.65	8.98	0.25	0.26	2.91	0.20	0.03	0.00	0.99	0
$\beta(2)$	2.59	1.22	2.14	1.80	3.69	0.94	0.17	2.03	0.10	0.01	0.00	2.07	0	0
$\beta(3)$	2.45	1.50	1.59	0.85	1.60	0.17	0.04	0.04	0.01	0.00	0.00	0.01	0	0

Cuadro 5.14: *cálculo de las β normalizadas por el factor* $N = \sum_{s'\in\sigma_i} \sum_{s\in\sigma_i} \gamma_i(s',s)\beta_i(s')$

Cálculo de la σ para los valores de entrada 0 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	1.39	0.99	0.31	3.11	0.03	1.365	6.38	0.01	0.38	0.85	2.49	0.00	0.00	0.01
$\sigma(1)$	0	0	1.57	0.25	0.66	0.172	0.04	0.33	0.01	0.01	0.01	0.01	0	0
$\sigma(2)$	0	0.51	0.08	0.20	0.56	0.020	0.00	0.02	0.00	0.00	0.00	0.06	0	0
$\sigma(3)$	0	0	0.17	0.03	0.25	0.544	0.01	0.13	0.16	0.01	0.00	0.00	0.15	0
Cálculo de la σ para los valores de entrada 1 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	2.76	0.13	0.22	0.26	2.11	0.34	0.04	0.60	.05	0.00	0.00	2.39	0	0
$\sigma(1)$	0	0	2.86	0.11	1.72	0.09	0.69	0.05	.03	1.66	0.76	0.00	0.00	1.48
$\sigma(2)$	0	2.68	0.12	1.51	0.01	2.94	0.10	0.02	.68	0.14	0.01	0.00	1.01	0
$\sigma(3)$	0	0	0.09	0.11	0.05	0.02	0.00	0.00	.00	0.00	0.00	0.00	0	0

Cuadro 5.15: *Valores σ para las distintas transiciones del trellis del decodificador 1 en la iteración 2*

$L(b_i/\mathbf{r}) = L(b_i)+L_c r_{i1}+L_e(b_i)$ =(1.7018, 1.5529, -0.3695, 0.3086, 0.9667, 0.3548, 1.1227, 4.4237, 4.1584, 2.8113, -2.5672, 4.9859, 5.2168, 7.4908)

L_{e1} = (0.6799, 0.6238, 0.4282, -0.5915, 0.9463, 0.4827, -2.0275, 0.3308, 0.3199, 0.7273, -1.1710, 3.3597, 1.1720, 1.1286)

Iteración 2, decodificador 2

Cálculo de la γ para los valores de entrada 0 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}$														
$\gamma(0,0)$	0.24	0.99	0.25	0.25	0.09	0.64	1.63	3.52	1.48	0.69	0.21	0.03	.34	0.52
$\gamma(1,2)$	0.24	0.99	0.25	0.25	0.09	0.64	1.63	3.52	1.48	0.69	0.21	0.03	.34	0.52
$\gamma(2,3)$	1.91	1.11	1.30	1.87	0.09	0.37	0.62	4.23	0.78	0.02	1.29	0.65	.54	0.60
$\gamma(3,1)$	1.91	1.11	1.30	1.87	0.09	0.37	0.62	4.23	0.78	0.02	1.29	0.65	.54	0.60
Cálculo de la γ para los valores de entrada 1 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}$														
$\gamma(0,2)$	4.04	1.00	3.92	3.95	10.02	1.54	.61	0.28	0.67	1.44	4.74	27.70	2.88	1.92
$\gamma(2,0)$	4.04	1.00	3.92	3.95	10.02	1.54	.61	0.28	0.67	1.44	4.74	27.70	2.88	1.92
$\gamma(3,1)$	0.52	0.90	0.76	0.53	10.07	2.65	.59	0.23	1.26	36.34	0.77	1.52	1.83	1.64
$\gamma(1,3)$	0.52	0.90	0.76	0.53	10.07	2.65	.59	0.23	1.26	36.34	0.77	1.52	1.83	1.64

Cuadro 5.16: *Valores γ para las distintas transiciones del trellis del decodificador 1 en la iteración 1*

Cálculo de la γ_e para los valores de entrada 0 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,0)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92
$\gamma_e(1,2)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92
$\gamma_e(2,3)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
$\gamma_e(3,1)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
Cálculo de la γ_e para los valores de entrada 1 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,2)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
$\gamma_e(2,0)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
$\gamma_e(3,1)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92
$\gamma_e(1,3)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92

Cuadro 5.17: *Valores γ_e para las distintas transiciones del trellis del decodificador 1 en la iteración 1*

Cálculo de $\alpha_{i-1}(s') = \frac{1}{N}\sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$															
$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\alpha(0)$	1	.05	0.02	0.55	0.27	0.08	0.38	0.36	0.32	0.27	0.02	0.58	0.25	0.04	0.42
$\alpha(1)$	0	0	0.42	0.23	0.08	0.58	0.30	0.17	0.15	0.23	0.41	0.21	0.00	0.51	0.24
$\alpha(2)$	0	.94	0.02	0.07	0.58	0.27	0.20	0.32	0.17	0.21	0.03	0.06	0.72	0.28	0.14
$\alpha(3)$	0	0	0.52	0.14	0.05	0.06	0.10	0.13	0.34	0.27	0.52	0.13	0.01	0.15	0.18

Cuadro 5.18: *Cálculo de las α normalizadas por el factor $N = \sum_{s'\in\sigma_{i-1}} \sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$*

Cálculo de $\beta_i(s') = \frac{1}{N}\sum_{s\in\sigma_i}\gamma_i(s',s)\beta_i(s')$														
$i=$	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\beta(0)$	0.29	0.95	0.32	0.24	0.08	0.39	0.56	.35	.00	0.00	.49	0.00	.1994	1
$\beta(1)$	0.29	0.70	0.33	0.23	0.08	0.21	0.06	.77	.00	0.12	.02	0.02	.7366	0
$\beta(2)$	0.23	0.30	0.46	0.23	0.09	0.41	0.10	.01	.16	0.00	.01	0.05	0	0
$\beta(3)$	0.24	0.31	0.26	0.69	0.26	0.96	1.47	.01	.04	0.00	.01	0.01	0	0

Cuadro 5.19: *cálculo de las β normalizadas por el factor $N = \sum_{s'\in\sigma_i}\sum_{s\in\sigma_i}\gamma_i(s',s)\beta_i(s')$*

Cálculo de la σ para los valores de entrada 0 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	0.10	0.05	0.00	0.04	0.02	0.04	0.35	.11	0.00	.01	0.00	0.00	0.04	0.04
$\sigma(1)$	0	0	0.08	0.02	0.00	0.31	0.05	.00	0.46	.00	0.00	0.00	0	0
$\sigma(2)$	0	0.31	0.01	0.13	0.15	0.20	0.18	.00	0.00	.00	0.00	0.00	0	0
$\sigma(3)$	0	0	0.39	0.09	0.00	0.00	0.00	.11	0.00	.00	0.03	0.01	0.01	0
Cálculo de la σ para los valores de entrada 1 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	0.64	0.01	0.03	0.35	0.02	0.02	0.02	0.00	0.51	0.00	.00	0.15	0	0
$\sigma(1)$	0	0	0.31	0.15	0.01	0.17	0.10	0.06	0.00	0.00	.50	0.00	0.00	0.55
$\sigma(2)$	0	0.62	0.00	0.00	0.05	0.07	0.02	0.23	0.00	0.13	.00	0.00	0.42	0
$\sigma(3)$	0	0	0.06	0.03	0.01	0.07	0.25	0.00	0.01	0.00	.00	0.00	0	0

Cuadro 5.20: *Valores σ para las distintas transiciones del trellis del decodificador 1 en la iteración 1*

$L(b_i/\mathbf{r}) = L(b_i)+L_c r_{i1}+L_e(b_i)$ = (2.5592, 0.4568, 0.8955, 1.3651, 3.9561, 0.9416, -0.4006, -2.4552, -0.0314, 5.7853, 3.7764, 5.6497, 3.7847, 3.7657)

L_{e1} =(1.8111, 0.5531, -0.2058, 0.6157, -0.6588, -0.4707, -0.3777, 0.2482, 0.1236, 1.8254, 2.4757, 1.9078, 5.2168, 7.4908)

$deinterleaver(L(b_i/\mathbf{r}))$ = (3.7764, 1.3651, -0.4006, 0.4568, 0.8955, -0.0314, 0.9416, 5.6497, 3.9561, 2.5592, -2.4552, 5.7853)

x' =(1, 1, -1, 1, 1, -1, 1, 1, 1, 1, -1, 1)

x =(1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1)

La posición 7 sigue siendo errónea pero la 6 ya se ha corregido en esta segunda iteración. Podemos ver que no se corrige la secuencia completamente hasta la iteración 4.

Iteración 4, decodificador 1

Cálculo de la γ para los valores de entrada 0 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}$															
$\gamma(0,0)$	0.06	1.41	1.64	1.18	0.23	2.04	0.44	0.01	0.19	0.33	6.34	0.04	0.01	0.00
$\gamma(1,2)$	0.06	1.41	1.64	1.18	0.23	2.04	0.44	0.01	0.19	0.33	6.34	0.04	0.01	0.00
$\gamma(2,3)$	0.07	0.29	1.87	0.22	2.88	0.75	0.14	0.06	0.13	0.24	0.62	1.25	0.00	0.01
$\gamma(3,1)$	0.07	0.29	1.87	0.22	2.88	0.75	0.14	0.06	0.13	0.24	0.62	1.25	0.00	0.01
Cálculo de la γ para los valores de entrada 1 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}$															
$\gamma(0,2)$	14.58	0.70	0.60	0.84	4.19	0.48	2.24	84.04	5.26	2.96	0.15	20.09	92.98	962.14
$\gamma(2,0)$	14.58	0.70	0.60	0.84	4.19	0.48	2.24	84.04	5.26	2.96	0.15	20.09	92.98	962.14
$\gamma(3,1)$	12.92	3.43	0.53	4.39	0.34	1.32	7.03	14.34	7.14	4.16	1.61	0.79	360.82	151.56
$\gamma(1,3)$	12.92	3.43	0.53	4.39	0.34	1.32	7.03	14.34	7.14	4.16	1.61	0.79	360.82	151.56

Cuadro 5.21: *Valores γ para las distintas transiciones del trellis del decodificador 1 en la iteración 4*

Cálculo de la γ_e para los valores de entrada 0 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,0)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	0.41	1.16	1.18	3.19	0.19	1.96	0.39
$\gamma_e(1,2)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	0.41	1.16	1.18	3.19	0.19	1.96	0.39
$\gamma_e(2,3)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	2.42	0.85	0.84	0.31	5.02	0.50	2.51
$\gamma_e(3,1)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	2.42	0.85	0.84	0.31	5.02	0.50	2.51
Cálculo de la γ_e para los valores de entrada 1 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,2)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	2.42	0.85	0.84	0.31	5.02	0.50	2.51
$\gamma_e(2,0)$	1.06	0.45	1.06	0.43	3.47	0.60	0.56	2.42	0.85	0.84	0.31	5.02	0.50	2.51
$\gamma_e(3,1)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	0.41	1.16	1.18	3.19	0.19	1.96	0.39
$\gamma_e(1,3)$	0.94	2.20	0.93	2.27	0.28	1.64	1.76	0.41	1.16	1.18	3.19	0.19	1.96	0.39

Cuadro 5.22: *Valores γ_e para las distintas transiciones del trellis del decodificador 1 en la iteración 4*

Cálculo de $\alpha_{i-1}(s') = \frac{1}{N}\sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$															
$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\alpha(0)$	1	0.00	0.00	0.24	0.09	0.76	0.64	0.12	0.62	0.12	.21	0.25	0.11	0.02	0.94
$\alpha(1)$	0	0	0.91	0.06	0.77	0.04	0.08	0.35	0.11	0.26	.58	0.04	0.07	0.80	0.00
$\alpha(2)$	0	0.99	0.00	0.66	0.07	0.13	0.19	0.38	0.22	0.55	.11	0.66	0.68	0.03	0.02
$\alpha(3)$	0	0	0.07	0.01	0.06	0.05	0.07	0.13	0.04	0.05	.09	0.03	0.11	0.13	0.02

Cuadro 5.23: *Cálculo de las α normalizadas por el factor $N = \sum_{s'\in\sigma_{i-1}} \sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$*

Cálculo de $\beta_i(s') = \frac{1}{N}\sum_{s\in\sigma_i}\gamma_i(s',s)\beta_i(s')$														
$i=$	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\beta(0)$	25	195	305	452	145	274	1405	1	49	865	541	0	0	1
$\beta(1)$	22	212	266	184	087	1292	40	5	432	21	1	0	3	0
$\beta(2)$	53	133	367	185	403	116	20	64	15	0	0	151		0
$\beta(3)$	42	161	188	87	234	17	5	1	1	0	0	0		0

Cuadro 5.24: *cálculo de las β normalizadas por el factor $N = \sum_{s'\in\sigma_i}\sum_{s\in\sigma_i}\gamma_i(s',s)\beta_i(s')$*

Cálculo de la σ para los valores de entrada 0 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	23	2	0.000.5	255.5	03.8	344.1	1.6	0.1	35.8	126.2	365.0	0	0	0
$\sigma(1)$	0	0	0.316.4	027.4	89.5	009.2	3.2	9.4	2.0	0.2	0.0	1.3	0	0
$\sigma(2)$	0	72	0.000.2	025.7	61.4	001.4	0.6	1.3	0.2	0.1	0.0	0	0	0
$\sigma(3)$	0	0	0.022.2	001.6	18.9	043.9	1.6	1.8	15.4	1.0	0.0	0	0.2	0
Cálculo de la σ para los valores de entrada 1 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	56.2	0.28	0.69	20.2	127	54.2	7.4	19.7	8.0	0.05	0	193	0	0
$\sigma(1)$	0	0	299	12.8	389	8	70.5	1.5	4.9	195	98.7	0	0.0	2.02
$\sigma(2)$	0	466	0.22	281	1.8	282	13.8	0.8	111	14	0.49	0	4.2	0
$\sigma(3)$	0	0	13.8	3.8	4.1	1.6	0.67	0.08	0.04	0.01	0	0	0	0

Cuadro 5.25: *Valores σ para las distintas transiciones del trellis del decodificador 1 en la iteración 4*

$L(b_i/\mathbf{r}) = L(b_i) + L_c r_{i1} + L_e(b_i) =$ (6.0987, 2.7159, -1.2023, 1.3397, 1.4758, -0.5804, -0.0999, 7.6606, 4.4753, 3.0119, -2.6724, 7.7764, 13.5534, 17.2723)

$L_{e1} =$(0.8595, 1.8314, -0.0771, 0.0247, 1.1042, -0.1418, -2.8605, 0.5657, 0.8487, 0.4996, -1.3020, 5.0024, 7.9321, 7.8851)

Iteración 4, decodificador 2

Cálculo de la γ para los valores de entrada 0 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}$															
$\gamma(0,0)$	0.27	0.72	0.23	0.13	0.07	0.98	2.10	3.76	2.02	0.30	0.19	0.03	0.01	0.01
$\gamma(1,2)$	0.27	0.72	0.23	0.13	0.07	0.98	2.10	3.76	2.02	0.30	0.19	0.03	0.01	0.01
$\gamma(2,3)$	2.14	0.81	1.20	1.02	0.07	0.57	0.80	4.52	1.07	0.01	1.18	0.58	0.01	0.02
$\gamma(3,1)$	2.14	0.81	1.20	1.02	0.07	0.57	0.80	4.52	1.07	0.01	1.18	0.58	0.01	0.02
Cálculo de la γ para los valores de entrada 1 $\gamma_i(s',s) = Ce^{b_i L(b_i)/2} e^{\frac{L_c}{2}\mathbf{r}_i \cdot \mathbf{x}_i}$															
$\gamma(0,2)$	3.61	1.37	4.25	7.24	13.06	1.02	0.47	0.26	0.49	3.28	5.19	31.16	84.71	56.35
$\gamma(2,0)$	3.61	1.37	4.25	7.24	13.06	1.02	0.47	0.26	0.49	3.28	5.19	31.16	84.71	56.35
$\gamma(3,1)$	0.46	1.22	0.82	0.97	13.12	1.74	1.24	0.22	0.92	82.63	0.84	1.71	53.99	48.13
$\gamma(1,3)$	0.46	1.22	0.82	0.97	13.12	1.74	1.24	0.22	0.92	82.63	0.84	1.71	53.99	48.13

Cuadro 5.26: *Valores γ para las distintas transiciones del trellis del decodificador 2 en la iteración 4*

Cálculo de la γ_e para los valores de entrada 0 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,0)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92
$\gamma_e(1,2)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92
$\gamma_e(2,3)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
$\gamma_e(3,1)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
Cálculo de la γ_e para los valores de entrada 1 $\gamma_i(s',s) = e^{\frac{L_c}{2}\sum_{i=2}^{n} r_i x_i}$														
$\gamma_e(0,2)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
$\gamma_e(2,0)$	2.78	1.05	2.26	2.72	0.99	0.76	0.61	1.09	0.72	0.19	2.47	4.26	1.25	1.08
$\gamma_e(3,1)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92
$\gamma_e(1,3)$	0.35	0.94	0.44	0.36	1.00	1.30	1.61	0.91	1.37	5.01	0.40	0.23	0.79	0.92

Cuadro 5.27: *Valores γ_e para las distintas transiciones del trellis del decodificador 2 en la iteración 4*

Cálculo de $\alpha_{i-1}(s') = \frac{1}{N}\sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$															
$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\alpha(0)$	1	0.07	0.02	0.68	0.17	0.02	0.39	0.38	0.35	0.35	0.01	0.70	0.16	0.00	0.77
$\alpha(1)$	0	0	0.55	0.14	0.02	0.76	0.15	0.22	0.18	0.20	0.51	0.13	0.00	0.75	0.20
$\alpha(2)$	0	0.92	0.04	0.06	0.77	0.17	0.37	0.21	0.22	0.24	0.03	0.05	0.82	0.23	0.01
$\alpha(3)$	0	0	0.37	0.10	0.02	0.03	0.07	0.16	0.23	0.20	0.42	0.10	0.00	0.00	0.01

Cuadro 5.28: *Cálculo de las α normalizadas por el factor $N = \sum_{s'\in\sigma_{i-1}} \sum_{s\in\sigma_{i-1}} \gamma_i(s',s)\alpha_{i-1}(s')$*

Cálculo de $\beta_i(s') = \frac{1}{N}\sum_{s\in\sigma_i}\gamma_i(s',s)\beta_i(s')$														
$i=$	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\beta(0)$	5.1	12.8	14.0	5.52	3.24	15.06	14.01	8.70	0.01	0.07	15.64	0.00	0.00	1
$\beta(1)$	6.0	29.8	11.8	6.62	3.21	6.72	1.04	35.61	0.11	2.09	0.01	0.00	0.95	0
$\beta(2)$	11.3	7.92	5.3	6.61	2.70	27.52	3.43	0.03	75.81	0.00	0.00	1.92	0	0
$\beta(3)$	9.1	9.12	5.3	11.93	5.83	41.80	68.91	0.03	0.02	0.00	0.00	0.00	0	0

Cuadro 5.29: *cálculo de las β normalizadas por el factor $N = \sum_{s'\in\sigma_i}\sum_{s\in\sigma_i}\gamma_i(s',s)\beta_i(s')$*

Cálculo de la σ para los valores de entrada 0 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	1.8	0.86	0.15	1.39	0.56	0.55	8.90	3.05	0.00	0.13	0.12	0.00	0.00	0.01
$\sigma(1)$	0	0	1.32	0.34	0.07	27.74	0.85	0.00	19.70	0.00	0.00	0.06	0	0
$\sigma(2)$	0	8.96	0.57	2.23	4.50	5.46	16.17	0.00	0.00	0.00	0.00	0.00	0	0
$\sigma(3)$	0	0	9.90	1.89	0.08	0.16	0.04	6.63	0.01	0.08	0.01	0.00	0.01	0
Cálculo de la σ para los valores de entrada 1 $\sigma_i(s) = \alpha_{i-1}(s')\gamma_{extr}(s',s)\beta_k(s)$														
$\sigma(0)$	31.5	0.59	0.30	12.33	0.46	0.58	0.83	0.01	19.40	0.00	0.00	5.79	0	0
$\sigma(1)$	0	0	17.7	2.11	0.08	8.85	1.33	2.16	0.00	0.00	20.14	0.00	0.00	0.81
$\sigma(2)$	0	26.17	0.24	0.16	2.48	1.50	0.63	7.08	0.03	2.56	0.00	0.00	0.62	0
$\sigma(3)$	0	0	0.87	0.46	0.15	1.71	8.22	0.00	0.00	0.00	0.00	0.00	0	0

Cuadro 5.30: *Valores σ para las distintas transiciones del trellis del decodificador 2 en la iteración 4*

Una vez tenemos todos los valores estamos en disposición de completar la ecuación 5.16 para encontrar los bits codificados por el codificador uno en la primera iteración:

$L(b_i/\mathbf{r}) = L(b_i) + L_c r_{i1} + L_e(b_i)$ =(3.3460, 1.5221, 1.7289, 2.9003, 4.6533, -0.4061, -1.3850, -2.8805, -0.7941, 8.0529, 6.4300, 8.4917, 12.6266, 12.5846)

L_{e1} =2.8257, 1.0022, 0.4697, 0.9434, -0.4905, -0.9854, -0.8567, -0.0461, -0.0145, 2.4504, 4.9497, 4.5150, 13.5534, 17.2723

$deinterleaver(L(b_i/\mathbf{r}))$ = (6.4300, 2.9003, -1.3850, 1.5221, 1.7289, -0.7941, -0.4061, 8.4917, 4.6533, 3.3460, -2.8805, 8.0529)

x' =(1, 1, -1, 1, 1, -1, -1, 1, 1, 1, -1, 1)

x =(1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1)

Como vemos en la cuarta iteración hemos recuperado el mensaje.

Capítulo 6

Detalles de la implementación

En este capítulo se expondrán los detalles implementados en Matlab para la realización del codificador-decodificador turbo y las simulaciones respectivas. Se han generado miles de líneas de código, horas de pruebas, rectificaciones y vuelta a empezar, no en vano la parte de diseño es la que más trabajo ha supuesto y en la que se ha centrado este proyecto. Lógicamente no vamos a incluir aquí todas las funciones diseñadas, más se intentarán aclarar los detalles más significativos y las funciones más relevantes del mejor modo posible.

6.1. Solución propuesta

Los códigos turbo forman parte de los códigos codificadores del canal, por lo que se utilizan para dar robusteza a una señal que atraviesa un canal afectado por una fuente de ruido, tal y como se muestra en la figura siguiente (fig. 6.1):

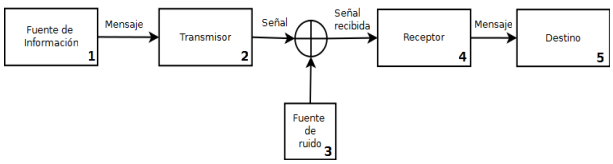


Figura 6.1: *Canal ruidoso de Shannon*

En nuestro caso de estudio, la correspondencia con el canal de transmisión de Shannon sería de la siguiente manera:

1. **La fuente de información**. En el esquema que queremos proponer la fuente de información serán las marcas digitales o watermarks. Las realizaremos en matlab, sin seguir patrón alguno para no incumplir la propiedad de ser indescifrable por métodos estadísticos como se había apuntado en el apartado 2.3, y también por ser de muy fácil implementación.

2. **Transmisor**. El transmisor de Shannon se corresponde a nuestro codificador turbo, como se ha comentado, se utilizan los turbocódigos por ser códigos muy próximos a la cota se Shannon, así será este punto uno de nuestros focos de diseño.

3. **Fuente de ruido** . Es este punto quizá el mas comprometido, puesto que la fuente de ruido a implementar no es más que el ataque por confabulación. Es por lo tanto una fuente de ruido "atípica", sin ruido gausiano.

4. **Receptor**. Nuestro receptor es el decodificador, es el punto que más posibilidades de diseño tiene, puesto que los turbocódigos se caracterizan por tener un codificador muy simple a costa de tener un decodificador ampliamente sofisticado.

5. **Destino**. Nuestro destino es un módulo que hemos denominado *matching filter* donde se procesará la señal recibida y donde se intentará relacionar con alguna de las posibles fuentes de origen. Esto es, las señales originales o usuarios deshonestos.

Una vez expuesta la idea de aplicación de códigos turbo como solución al borrado o manipulación de marcas digitales mediante ataques de confabulación, pasaremos a esquematizar el diseño implementado en matlab.

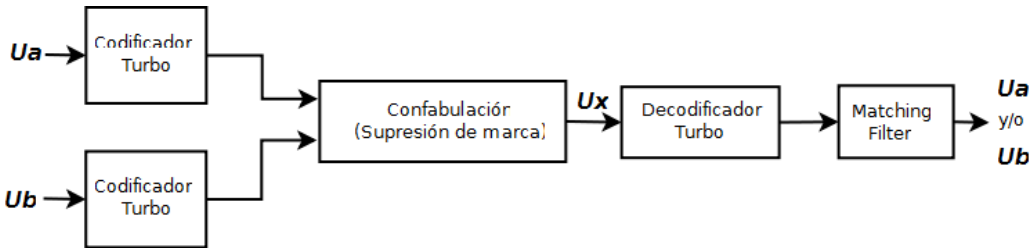


Figura 6.2: *Esquema del diseño a implemantar*

Una vez obtenido el esquema principal de simulación definiremos qué partes serán objeto de diseño en este proyecto:

- Usuarios: Los usuarios, es decir, los propietarios de la marca o código serán tratados como mensajes, es decir, como información no manipulada. Esta información representa en sí la marca digital embebida en una imagen o medio digital susceptible de manipulación. Es muy importante tener en cuenta la naturaleza de estas marcas, adaptándolas lo máximo posible a una distribución de copyright real, así tendremos en cuenta los siguientes aspectos:
  - La longitud del mensaje:el tamaño de la imagen debe ser acorde al medio que se quiere marcar, por ejemplo no es lo mismo marcar un video que una imagen. Además hay que tener en cuenta que las marcas no pueden tener longitud variable, pues es el decodificador turbo el que nos devolverá un código acorde con el tamaño que esté diseñado.
  - Número de confabuladores: Este parámetro tiene mucha importancia en el cálculo de probabilidades tal y como se verá más adelante, pues un conjunto de infinitos confabuladores o usuarios honestos no es real, y el objetivo de estudio de este proyecto és el de hallar los confabuladores dentro de un conjunto de posibles atacantes con marcas legales.
- La codificaciónr turbo: Es el núcleo del proyecto y en el que sea han dedicado más esfuerzos, entre otras cosas por los siguientes aspectos:
  - El turbocodificador se ha de diseñar de forma abierta no pudiéndose utilizar aplicativos cerrados. De esta forma se pueden hacer

modificaciones en cualquier parte del código y analizar las posibles variaciones y mejoras en el diseño, insertando otros medios de codificación si fuera necesario.

- Se debe testear el codificador diseñado antes de darlo por válido, ya que un codificador turbo de bajas prestaciones o mal diseñado invalidaría por completo el resto del estudio.
- Probar variaciones de diseño, como añadir más codificadores *RSC* y su topología -paralelo o serie-, diferentes tipos de *interleavers*, variables como la L_c, decisión soft o hard . . .

- Confabulación: aquí simularemos que por comparación los posibles atacantes han extraído las marcas legales, y que se disponen a sacar una marca falsa a partir de la combinación de estas. En este proyecto se simularán los siguientes casos

  - En cada posición elegimos aleatoriamente un usuario o marca y nos quedamos con su bit en esa posición. En el caso de haber solamente dos confabuladores las diferencias se deciden al 50 %.
  - Comparamos palabras de 3 bits de longitud de cada usuario, nos quedamos aleatoriamente con una de ellas.

  Cabe destacar que no se ha diseñado un algoritmo de elección de usuarios, sino que se realiza la elección aleatoriamente. De este modo la probabilidad en la elección de un bit o una palabra en concreto la marcará la estadística de la señal, por ejemplo, si confabulan diez usuarios donde en una posición concreta siete de ellos tienen un 1 y tres de ellos un 0, la marca confabulada tendrá un 1 con una probabilidad del 70 %.

- Cálculo de resultados: tan importante como el diseño del codificador de la marca es el análisis de los resultados. Claro está que será prácticamente imposible llegar obtener la marca original con un 100 % de fiabilidad, pues la naturaleza de la manipulación de los mensajes no lo permite y la degradación de las marcas es muy elevada. Así, debemos tratar los resultados de la forma que más nos acerque a los propósitos esperados. El método utilizado para identificar a los atacantes, será mediante la correlación de la marca una vez pasada por el proceso de decodificación con el resto de marcas conocidas y legales. Lo que se pretende analizar es si los resultados de las correlaciones con las marcas deshonestas serán lo suficientemente distantes respecto a los usuarios honestos, que no han participado en el fraude, como para asegurarse el éxito de la identificación sin llegar a falsos culpables.

6.2. Función turbocod.m

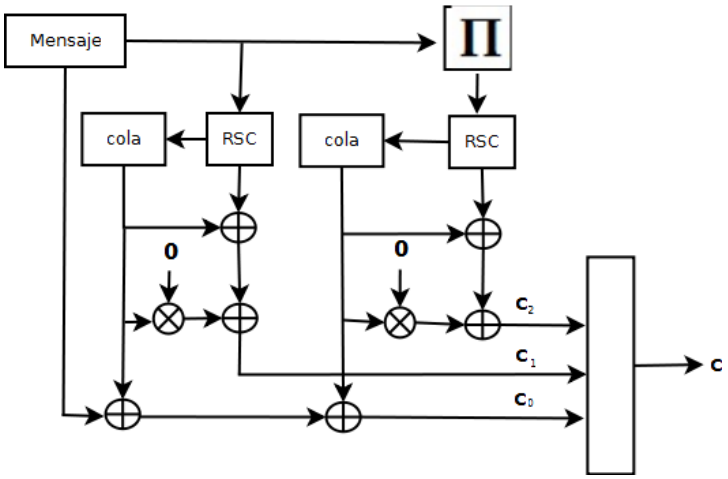


Figura 6.3: *Diagrama de estados de la función turbocod.*

Esta función que tiene como parámetros de entrada la señal a codificar, el trellis utilizado por los codificadores RSC y un interleaver, implementa el codificador turbo de la figura 5.1, aunque con una modificación para hacer el código terminado, esto es, que termine siempre en el estado del trellis 0. Esta modificación en realidad hace que la codificación turbo ya no sea una codificación continua si no por bloques de datos, que terminan y empiezan en el estado 0 del trellis. Estos bloques de datos en su mayoría serán el mensaje más unos bits de cola. La función que termina el código turbo es la función cola (linea 11). Esta función, que se puede ver en el anexo C.1, crea un grafo del trellis e implementa el algoritmo de Dijkstra para encontrar el camino que con menos saltos llega al estado 0.

Así el cuerpo central de la función se centra en hacer la codificación convolucional del mensaje, añadir los bits de cola, volver a codificar el mensaje (ahora con los bits de cola añadidos) y comprobar que efectivamente el estado final es el 0. Si volvemos a la figura 5.1 vemos que este proceso se ha de hacer de nuevo para las entradas entrelazadas, en este caso de codificador turbo PCCC entrelazamos la secuencia (línea 22), y tratamos la secuencia entralazada como una secuencia nueva a la que se le ha de volver a calcular los bits de cola.

Finalmente, para enviar la palabra código de ratio $\frac{1}{3}$ al canal, debemos "ordenar" la salida, para ello añadimos los bits de cola a la salida sistemática (mensaje original), y añadimos zeros a las dos salidas codificadas para igualar así el tamaño de las secuencias, estos bits serán descartados en la decodificación. Para finalizar, se realiza un entralazado matricial a la secuencia de salida que nos dará protección frente a ruido a ráfagas, y se envían los datos en forma bipolar.

```
1   function [output,interleaver]=turbocod(x,trellis,interleaver)
2
3   %[output,interleaver]=turbocod(x,trellis,interleaver)
4   % Turbo-codificador
5   %   x entrada
6   %   trellis codificadores RSC
7   %   interleaver interleaver structure
8
9
10  [outputa estatfinal]=convenc(x,trellis);
11  tail1=cola(trellis,estatfinal);
12
13  x1=horzcat(x,tail1);
14  [outputa estatfinal]=convenc(x1,trellis);
15  sortida1=outputa(2:2:length(outputa));%bits paridad codificador 1
16
17  if(estatfinal)
18      disp('error codigo no terminado');
19  end
20
21
22  x2=x(interleaver); %entrelazado antes de añadir los bits de cola
23
24  [outputb finalstate]=convenc(x2,trellis);
25  tail2=cola(trellis,finalstate);
26
27  x2=horzcat(x2,tail2);
28
29  [outputb finalstate]=convenc(x2,trellis);
30
31  sortida2=outputb(2:2:length(outputb));%bits paridad codificador 2
32
33  if(finalstate)
34      disp('error codi no acabat');
35  end
36
37  %añadimos bits de cola al mensaje original
38  x=horzcat(x,tail1,tail2);
39  sortida1=horzcat(sortida1,zeros(1,length(tail1)));
40  sortida2=horzcat(sortida2,zeros(1,length(tail1)));
41
42
43  output=[x sortida1 sortida2]; %mensaje codificado según
44                                  la estructura (x,c1,c2)
45
46  output=matintrlv(output,3,length(x));%protección contra el canal
47
48   output=2*output-ones(1,length(output));
```

6.3. Función decturbo.m

Esta función es la que nos implementa la decodificación turbo de las figuras 5.5 y 5.6, tiene como parámetros de entrada la señal procedente del canal, la constante de canal, el trellis utilizado por los codificadores RSC en codificación, el interleaver, también utilizado en el proceso de codificación, y el número de iteraciones que se desean realizar para la decodificación. Como salida tenemos los bits en formato hard (ceros y unos) y soft, valores reales.

La función tiene dos grandes bloques, el primero es el tratamiento de la señal de entrada, esto es, organizar los bis que llegan entrelazados del canal. Este proceso es necesario ya que en el proceso de decodificación cada decodificador constituyente (los que implementan el algoritmo BCJR), tienen su propia entrada, es decir, los bits de paridad del decodificador 1 no sirven para el decodificador 2 y viceversa. Así mismo, la entrada sistemática del codificador turbo es ligeramente distinta también para los dos decodificadores, ya que los bits de paridad también son distintos. Para el caso del decodificador 2, que ha de pasar a través de un desentrelazador, se tiene que procurar que los bits de cola no sean desentrelazados, ya que estos no pasaron por el interleaver en codificación, para solventar esto también es necesario modificar el $de-interleaver$ para que no modifiquen los bits de cola. Una vez hecho todo esto nos quedan cuatro secuencias agrupadas dos a dos, dos de ellas servirán para la decodificación en el primer decodificador y las otras dos en el segundo. En la figura 6.4 se muestra un esquema del proceso de organización de los bits a la entrada de la función decturbo.m, los números identifican las líneas de código en la función.

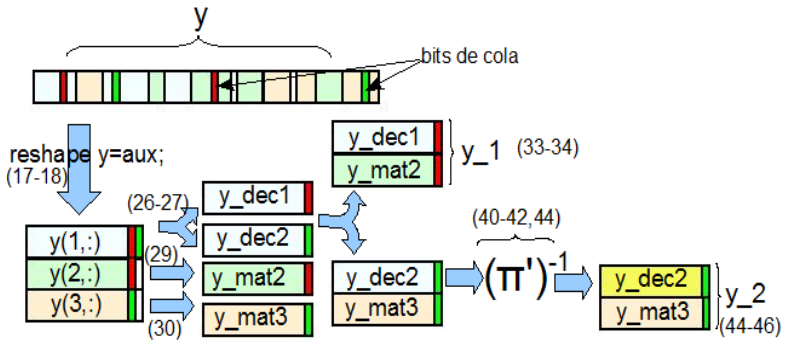


Figura 6.4: *Esquema del tratamiento de los bits de entrada por la función decturbo.*

```
1   function [L Lu] = decturbo( y, Lc, trellis, inter,itera)
2   %function [L Lu] = decturbo( y, Lc, trellis, inter,itera)
3   %
4   % L, bits soft de salida,
5   % Lu, bits hard salida,
6   % y, secuencia entrada,
7   % trellis, trellis del turbocodificador convolucional,
8   % Lc, constant de canal
9   % inter, interleaver del segundo cc,
10  % itera, nombre de iteracines del decodificador turbo
11
12  tailength=log2(trellis.numStates);%longitud bits de cola
13  long=numel(y)/3; %longitud del codigo que recibimos
14
15
16  % Cuantificamos la entrada y calculamos el deinterleaver
17  aux=reshape(y,3,numel(y)/3);
18  y=aux;
19  tail1=y(1,long-(2*tailength-1):long-tailength);
20  tail2=y(1,long-(tailength-1):long);
21  %la primera fila de la matrz no sirve para los
22  %dos decodificadores, ya que tienen bits de cola distintos,
23  %que no han pasado por el codificador.
24  %así renombramos estos vectores.
25
26  y_dec1=y(1,1:(long-tailength));
27  y_dec2=horzcat(y(1,1:(long-2*tailength)),tail2);
28
29  y_mat2=y(2,1:length(y)-tailength);%al decodificador dos no pasamos
30                                    los bits de cola del decodificador 1
31  y_mat3=y(3,1:length(y)-tailength);
32
33  y_1(1,:)=y_dec1;
34  y_1(2,:)=y_mat2;
35
36  [aux, deinter]=sort(inter);
37  clear aux;
38
39  %así porque los bits de cola no se entrelazan
40  deinter_tail=(length(deinter)+1):1:(length(inter)+tailength);
41  deinter=horzcat(deinter,deinter_tail);
42  inter=horzcat(inter,deinter_tail);
43
44  y_2(1,:)=y_dec2(inter);%este lo tenemos que interleavear
45                         ya que va al decodificador dos.
46  y_2(2,:)=y_mat3;
```

El segundo bloque, que se presenata en la siguiente figura (6.5), nos muestra un esquema de una iteración de decodificación implementada mediante un bucle while en la función decturbo.m, mostrada en la página siguiente. La diferencia respecto a la figura 5.6 es el tratamiento que se le hace a los bits de cola.

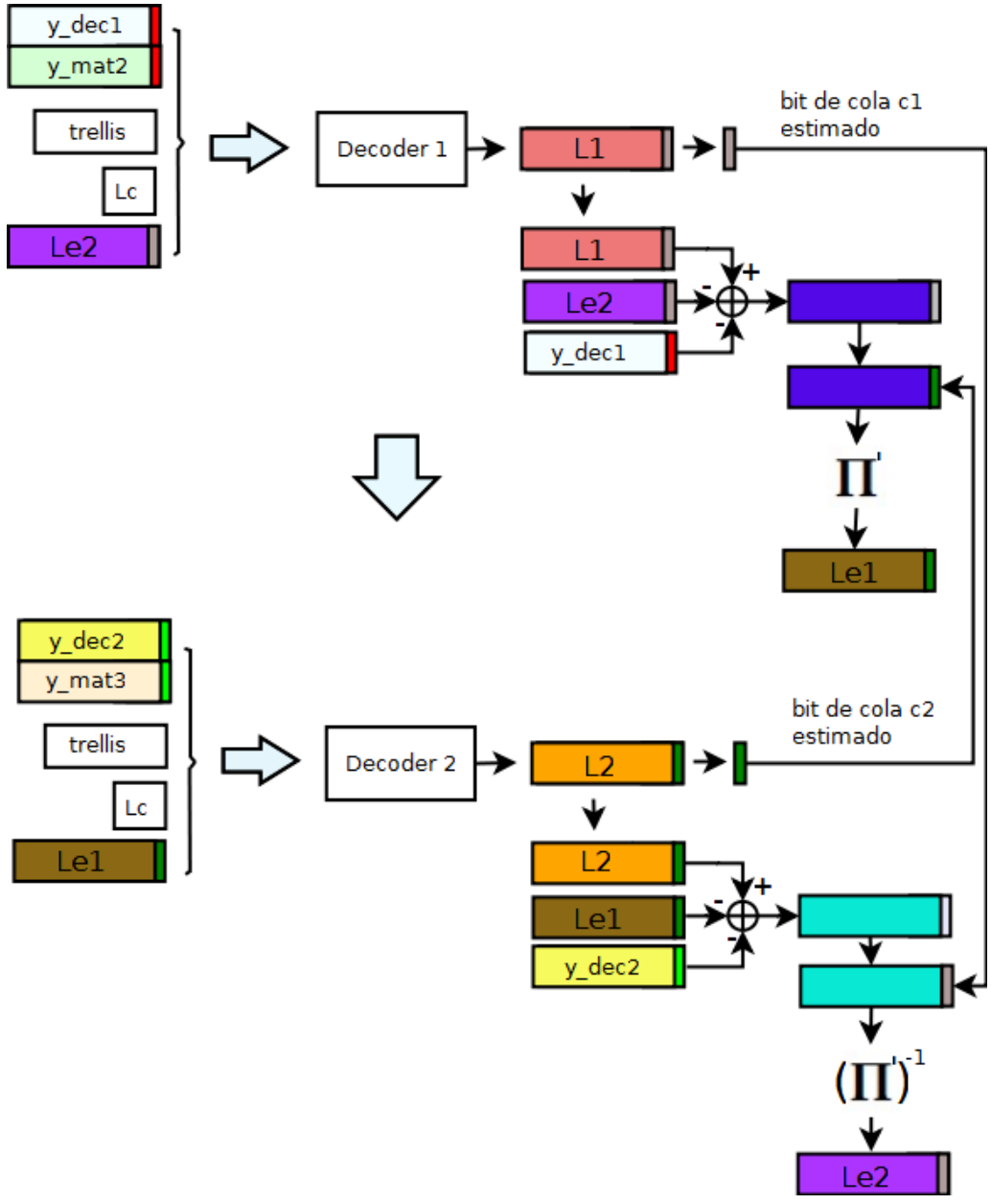


Figura 6.5: *Diagrama de estados de la función turbocod.*

```
48  %   Iniciem decodificació
49  Le1=zeros(numel(y_mat3),1)';
50  Le2=zeros(numel(y_mat3),1)';
51  L1=zeros(numel(y_mat3),1);
52  L2=zeros(numel(y_mat3),1);
53  cua1_estimada=0*tail1;
54  cua2_estimada=0*tail2;
55  Luk=ones(numel(y_mat3),1);
56
57
58  for i=1:itera
59     % pause;
60      [L1 Lu]=maplog(y_1,trellis,Lc,Le2(deinter));
61      %cua1_estimada=bits de cua1 estimados, no han de pasar
62      al decodificador 2 ya que son diferentes
63      cua1_estimada=L1((length(L1)-length(tail1)+1):length(L1));
64
65      Le1=L1-Le2(deinter) -y_dec1;%
66      Le1=Le1(1:length(Le1)-length(tail1));
67      Le1=horzcat(Le1,cua2_estimada);
68      L1=L1(1:(length(L1)-length(tail1)));%eliminamos los bits
69                                          de cola del codificador 1
70      L1=horzcat(L1,cua2_estimada); %añadimos los bits de
71                                    cola del codificador 2
72
73      [L2 Lu]=maplog(y_2,trellis,Lc,Le1(inter));
74
75      %cua2_estimada=bits de cola de 2
76      cua2_estimada=L2((length(L2)-length(tail2)+1):length(L2));
77
78      Le2=L2-Le1(inter)-y_dec2(inter);
79
80      Le2=Le2(1:length(Le2)-length(tail1));
81      Le2=horzcat(Le2,cua1_estimada);
82
83      L2=L2(1:(length(L2)-length(tail1)));%eliminamos los bits
84                                          de cola del codificador 2
85      L2=horzcat(L2,cua1_estimada); %añadimos bits
86                                  de cola del codificador 1
87   end
88   L=L2(deinter);
89   L=L(1:length(Lu)-tailength);
90   Lu=Lu(deinter);
91
92   Lu=Lu(1:length(Lu)-tailength);
93   Lu=(Lu+1)/2;
```

6.4. Función maplog.m

La función maplog.m nos implementa el algoritmo log-BCJR detallado en el capítulo 5, y de manera más detallada, en la sección 5.2.3 se muestran los pasos del algoritmo mediante un ejemplo, por ello a continuación expondremos el diseño en lenguaje Matlab implementado sin necesidad de extendernos demasiado en los detalles para no alargarnos en exceso. Se desglosa el código en partes para hacerlo lo más claro posible.

En la primera parte de la función disponemos los datos de forma adecuada, para ello colocamos en forma bipolar la salidas del trellis tanto para la entrada del bit 0 como la entrada del bit 1, esto es, modificamos los valores de trellis.output que se puede ver en la figura 4.11.

```
1  function [Le Lu]=maplog(y,trellis,Lc,La)
2  % Le, bits de salida soft,
3  % Lu, bits de salida hard,
4  % y, bits entrada
5  % trellis, trellis del turbocodifcador,
6  % Lc, constante de canal
7  % La, información a priori
8
9  trellis.nextStates=trellis.nextStates+1;
10 % nueva funcion logmap, deve ser extensible a cualquier tipo de
11 % turbocodificador, independientemente del número
12 %de codificadores que tenga.
13 inf=300000000000;
14 long=length(y);
15 %disponemos los los siguentes estados del siguiente modo.
16 %de este modo la primera fila corresponde al bit de salida -1
17 %la segunda al bit de salida 1. El número de columna indica
18 %el estado actual, y el numro dentro de esa columna
19 %es el estado siguiente.
20
21 proxState(1,:)=trellis.nextStates(:,1);
22 proxState(2,:)=trellis.nextStates(:,2);
```

```
24 x_0=ones(trellis.numStates,2);%salida para el bit de entrada 0
25 x_1=ones(trellis.numStates,2);%salida para el bit de entrada 1
26 for i=1:trellis.numStates
27     switch trellis.outputs(i,1)
28         case 0;
29             x_0(i,1)=-1;
30             x_0(i,2)=-1;
31         case 1;
32             x_0(i,1)=-1;
33             x_0(i,2)=1;
34         case 2;
35             x_0(i,1)=1;
36             x_0(i,2)=-1;
37         case 3;
38             x_0(i,1)=1;
39             x_0(i,2)=1;
40     end
41
42     switch trellis.outputs(i,2)
43         case 0;
44             x_1(i,1)=-1;
45             x_1(i,2)=-1;
46         case 1;
47             x_1(i,1)=-1;
48             x_1(i,2)=1;
49         case 2;
50             x_1(i,1)=1;
51             x_1(i,2)=-1;
52         case 3;
53             x_1(i,1)=1;
54             x_1(i,2)=1;
55     end
56 end
57 y_s=y(1,:); % bits secuenciales
58 y_p=y(2,:); % bits de paridad
```

Calculamos los coeficientes γ mediante la ecuación 5.15 y la γ extrínseca, quedándonos tan sólo con los bits de paridad.

```
70  %% Calculamos la gamma extrínseca
71  %la llegada será de la siguiente forma
72
73  %%inicializamos gamma, para la salida 1 y para la salida -1
74  gamma_1=zeros(trellis.numStates,long);
75  gamma_0=zeros(trellis.numStates,long);
76
77  for i=1:long
78      for j=1:trellis.numStates
79
80         % gamma_0(j,i)=0.5*Lc*(y_s(i)*x_0(j,1)+y_p(i)*x_0(j,2));
81          %gamma_1(j,i)=0.5*Lc*(y_s(i)*x_1(j,1)+y_p(i)*x_1(j,2));
82          %gamma_0(j,i)=exp(gamma_0(j,i));
83          %gamma_1(j,i)=exp(gamma_1(j,i));
84
85
86          factor_0=exp(0.5*(La(i)*x_0(j,1)+Lc*y_s(i)*x_0(j,1)));%
87          factor_1=exp(0.5*(La(i)*x_1(j,1)+Lc*y_s(i)*x_1(j,1)));%
88          % ahora la gamma extrinseca. para la gamma extrinseca
89          % solo nos quedamos con los bits de paridad.
90          gammae_0(j,i)=0.5*Lc*(y_p(i)*x_0(j,2));
91          gammae_1(j,i)=0.5*Lc*(y_p(i)*x_1(j,2));
92          gammae_0(j,i)=exp(gammae_0(j,i));
93          gammae_1(j,i)=exp(gammae_1(j,i));
94          gamma_0(j,i)=factor_0*gammae_0(j,i);
95          gamma_1(j,i)=factor_1*gammae_1(j,i);
96      end
97  end
```

Una vez obtenidas las γ calculamos las métricas α, ecuación 5.8, para ello hemos de ir buscando a cada iteración del trellis que estados preceden al nuestro, una vez tenemos los valores α los normalizamos dividiéndolos por la suma de ellos en cada iteración del trellis.

```
106   %% cálculo de las alfas
107
108   alfa=zeros(trellis.numStates,long+1);
109   alfa(1,1)=1; % inicializamos alfa
110   alfan=alfa;
111   normalizer=ones(1,long+1);
112   for i=1:long
113     for state=1:trellis.numStates
114       %encuentro que posibles estados precedn al mio
115       prevstate_0=find(proxState(1,:)==state);
116       prevstate_1=find(proxState(2,:)==state);
117       alfa(state,i+1)=gamma_0(prevstate_0,i)*alfan(prevstate_0,i) ...
118                             +gamma_1(prevstate_1,i)*alfan(prevstate_1,i);
119       normalizer(i+1)=sum(alfa(:,i+1));
120       %obtenemos la gamma normalizada, que es la que necesitamos.
121       alfan(:,i+1)=alfa(:,i+1)/normalizer(i+1);
122     end
123   end
```

Hacemos el proceso análogo para obtener las métricas β (ecuación 5.9). También normalizamos los valores del mismo modo que se hizo en el cálculo de α.

```
126 %% cálculo de las betas
127
128 beta=zeros(trellis.numStates,long);
129 beta(1,long)=1;
130 betan=beta;
131 normalizer=normalizer(2:length(normalizer));
132 normalizer(1,long)=1;
133 for i=long:-1:2
134     for state=1:trellis.numStates %ojo aquí no tengo que buscar
135         prevstate_0=proxState(1,state);
136         prevstate_1=proxState(2,state);
137         beta(state,i-1)=gamma_0(state,i)*betan(prevstate_0,i) ...
138                         +gamma_1(state,i)*betan(prevstate_1,i);
139         betan(:,i-1)=beta(:,i-1)/normalizer(i-1);
140     end
141 end
```

Hacemos el cálculo de coeficientes σ, es decir, el denominador y numerador de la ecuación 5.17, el cual se obtiene a partir de las variables obtenidas anteriormente.

```
147  %% cálculo de las sigmas
148
149 sigma_0=zeros(trellis.numStates,long);
150 sigma_1=sigma_0;
151 alfanova=alfan(:,1:length(alfan)-1);
152 %Aquí realizamos lo siguiente alfan(s')*gamma(s',s)*betan(s)
153 % Ahora debemos encontrar el orden eso se mira en el trellis.
154 % En los valores de prox state son los siguientes mientras que
155 % las columnas son los actuales.
156
157 for i=1:trellis.numStates
158 sigma_0(i,:)=alfanova(i,:).*gammae_0(i,:).*betan(proxState(1,i),:);
159 sigma_1(i,:)=alfanova(i,:).*gammae_1(i,:).*betan(proxState(2,i),:);
160 end
```

Finalmente se estima la secuencia de bits, y se pasan los valores del decodificador en forma soft y hard, para esta última nos quedamos con el signo. Como hemos sacado los valores extrínsecos(ecuación 5.17) tan solo debemos sumar el resto de variables para obtener $L(b_i/\mathbf{r})$ (ecuación 5.18).

```
163 %% Estimamos la secuencia de bits
164 for i=1:long
165     sum_0=sum(sigma_0(:,i));
166     sum_1=sum(sigma_1(:,i)) ;
167     L(i)=sum_1/sum_0;
168     if(sum_0==0)
169           L(i)=inf;
170       end
171     if(sum_1==0);
172           L(i)=-inf;
173     end
174  end
175 pr=log(L);
176 Le=log(L)+La+Lc*y(1,:);
177 Lu=sign(Le);
```

Con la función maplog.m se termina la parte de diseño de la codificación turbo. A partir de este momento se ha trabajado en el diseño de la estructura de pruebas y simulaciones, las cuales deberán acercarnos a un posible caso de confabulación para la extracción de las marcas en un contexto real. No detallaremos aquí el código de todas las funciones implementadas, algunas de ellas no obstante se pueden consultar en el anexo C, de todos modos se mostrará un esquema para clarificar en la medida de lo posible la línea de simulaciones seguida.

6.5. Implementación de las simulaciones

Como se mostró en el capítulo anterior, el esquema básico de simulación se asemejaría al modelo de transmisión de señal de Shannon. Así, podríamos diferenciar tres partes básicas en el diseño del código; la generación de los usuarios, función *gencodi.m*, los tipos de confabulación para la extracción de la marca, función *comfabular.m* y la decodificación del usuario deshonesto, función *testfinal.m*.

- **gencodi.m:** Esta función de Matlab nos genera un conjunto de usuarios a los que se les aplica una codificación turbo mediante un trellis y un interleaver, que pasamos como parámetros de entradas.
- **comfabular.m** Esta función nos compara bit a bit los códigos de los usuarios que deseemos que confabulen y nos extrae un codigo resultante, los ataques pueden ser a nivel de bit o de palabra.
- **testfinal.m** Aquí básicamente se decodifica el usuario fraudulento y se realiza una posible correlación con los usuarios posibles, quedándonos con los picos en 0, los que pasen de cierto umbral son considerados usuarios deshonestos. El factor de diseño que se impone en este punto es que haya el menor número de falsos positivos, es decir, siempre intentaremos ajustar los parámetros de este filtro para que el número de errores en la decisión final sean mínimos. Este aspecto, como es lógico, siempre irá en detrimento de que posibles confabuladores no sean detectados.

La figura siguiente ejemplifica el proceso de simulación diseñado.

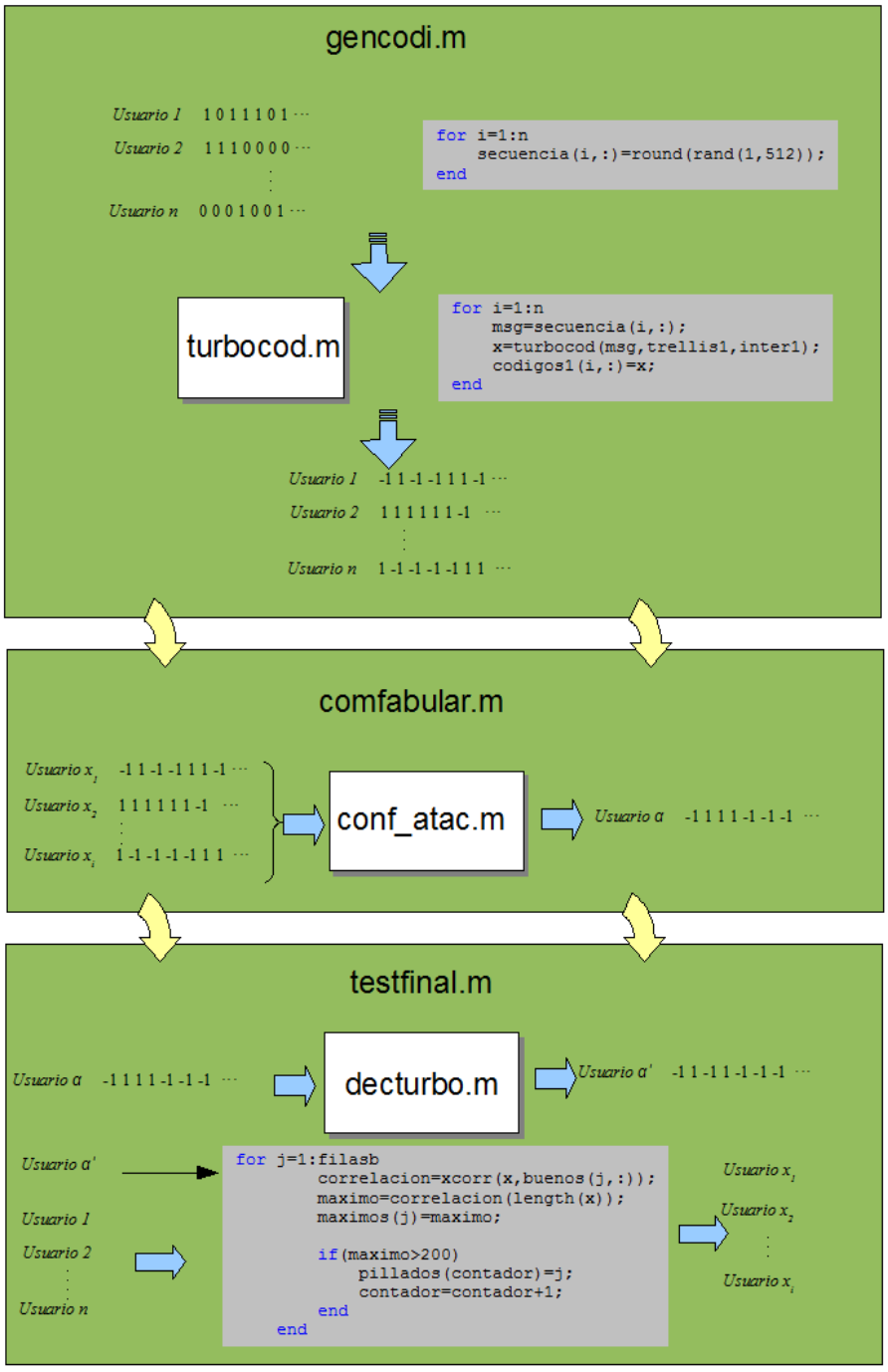


Figura 6.6: *Esquema del diseño de simulación propuesto*

Capítulo 7

Resultados

En esta sección mostraremos los resultados de las simulaciones realizadas. Para ello se han habilitado en Matlab varias funciones que realizan este cálculo. El principio básico de simulación es el comentado en el tema 5, en el que se intentará simular un canal de transmisión de Shannon donde se generarán un conjunto de usuarios de los cuales algunos de ellos confabularán para obtener una marca falsa. Una vez que en recepción recibamos todas las señales, se realizará una decodificación turbo que implementará lo que hemos denominado $''matching\ filter''$, el cual nos deberá reportar con la mayor fiabilidad posible alguno de los usuarios deshonestos que han participado en la generación de la marca falsa. El esquema de simulación se corresponde con el esquema 6.6, en el cual, si nos fijamos bien en la función testfinal.m veremos que consideramos como fraudulentos los usuarios que superan un valor umbral, que en el caso de la figura es 200. La salida por defecto del turbo decodificador es una salida $soft$, es decir, valores absolutos sin pasar por un decisor. Esto se ha hecho para evitar el sesgo de información que siempre lleva consigo un decisor. Por ejemplo que el decodificador nos reporte -60 en una posición concreta es lo mismo que nos reporte $-0{,}1$, aunque a priori este último valor es -1 con una fiabilidad menor que el caso de -60. Por ello, como el $matching\ filter$ consiste básicamente en un correlador que nos calculará el parecido entre dos marcas, no es necesario utilizar decisión $hard$, es decir ceros y unos.

De este modo, para facilitar el seguimiento de las pruebas, hemos dividido este tema en secciones correspondientes al número de confabuladores.

7.1. Confabulación por comparacion en bloques de tres bits

En este primer caso el método de confabulación entre usuarios consiste en comparar las secuencias en porciones de tres bits. Así, en el caso de dos confabuladores, se comparan las tríadas de bits y elegimos aleatoriamente una de ellas en el caso que difieran. Por ejemplo para el caso de 4 y 6 confabuladores elegimos aleatoriamente una tríada de cada usuario. Se han realizado múltiples simulaciones para poder adaptar mejor las variables propias de los códigos turbo, como pueden ser el número de iteraciones, la constante de canal o el trellis utilizado en los RSC, así como el umbral de corte a partir del cual consideraremos que una marca decodificada coincide lo suficiente como para identificarla como deshonesta. Cabe destacar que el principal parámetro a tener en cuenta es que no haya falsos positivos, de modo que siempre es mejor no capturar a ningún usuario que ha participado en la supresión de la marca que acusar a un usuario inocente.

7.1.1. Detección de dos confabuladores

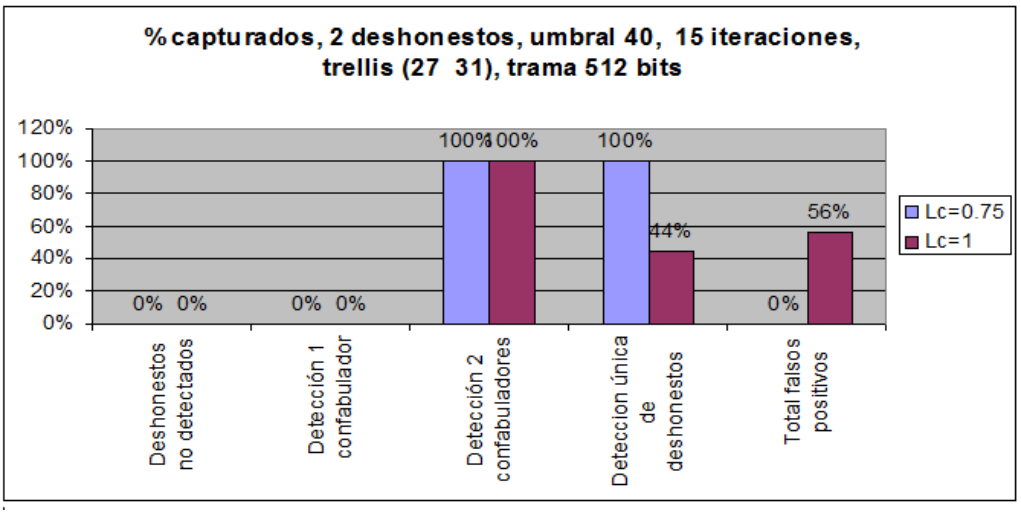


Figura 7.1: *% de captura de usuarios deshonestos generados con codificadores turbo de RSC* $(27, 31)_8$ *y constantes de canal Lc=0.75, Lc=1.*

Si nos fijamos en la primera simulación representada en la figura 7.1 observaremos que hay un 100 % de posibilidades de encontrar ambos confabuladores con tan sólo 15 iteraciones utilizando una constante de canal de 0.75. Por el contrario, si le damos más confianza a la secuencia derivada del canal observamos que los resultados no son tan buenos y el número de errores se eleva a más de la mitad. Este hecho se debe muy probablemente a que el umbral utilizado es demasiado pequeño para el orden de magnitud que alcanzan los bits decodificados con una constante de canal de 1.

7.1.2. Detección de 4 confabuladores

En este caso el número de confabuladores es de 4, por ello la señal resultante difiere en mayor medida de la señal original que en el caso de tan sólo dos usuarios deshonestos. La gráfica siguiente (fig. 7.2) muestra las simulaciones realizadas con RSC $(27\ 31)_8$ con constante de canal de 0.75 para distintos umbrales y número de iteraciones en el proceso de decodificación turbo.

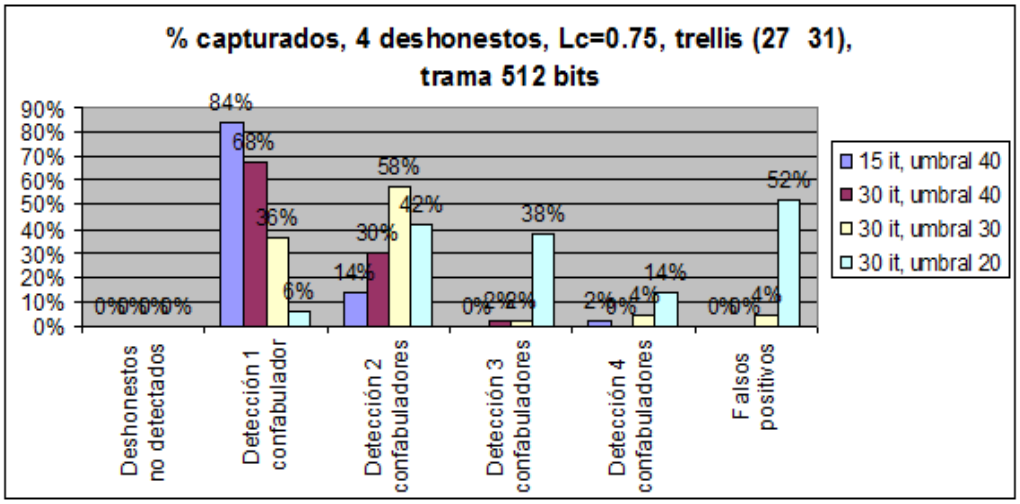


Figura 7.2: *% de captura de usuarios deshonestos generados con codificadores turbo de RSC* $(27, 31)_8$

Como se puede observar en la gráfica 7.2, la mejor opción la obtenemos con un umbral de 40 y 30 iteraciones, es este el caso en el que todavía no

hallamos falsos positivos y la probabilidad de encontrar a más de un usuario deshonesto es mayor. Como cabía esperar, al tener que rebajar el umbral de decisión, nos encontramos con que usuarios que no han participado de la confabulación son tratados como "culpables". Este hecho se puede deber tanto a la decodificación errónea de la señal como al parecido de las señales de forma fortuita, ya que han sido generadas de forma aleatoria. Cabe destacar también que el hecho de pasar de 15 a 30 iteraciones nos provoca que en pruebas donde antes tan solo detectábamos un confabulador, pasamos a detectar dos de ellos.

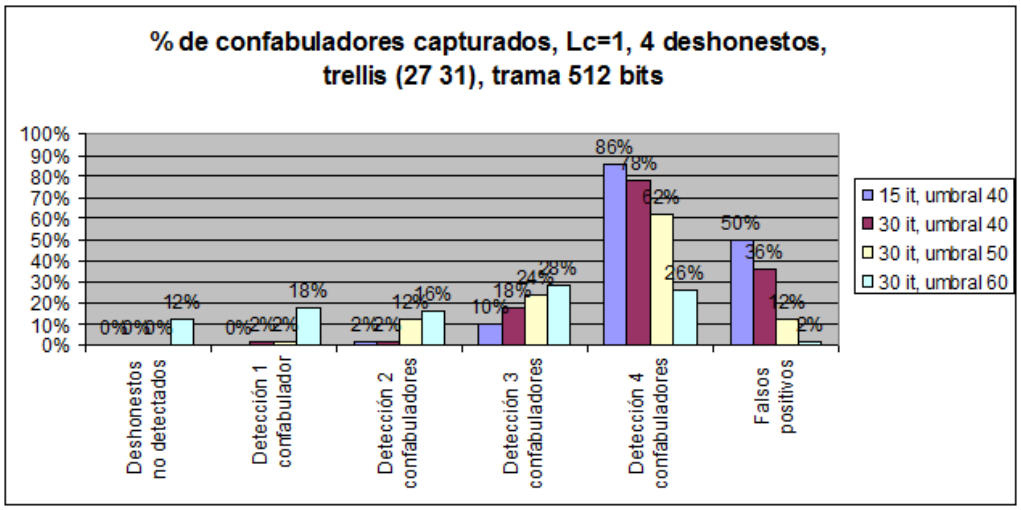
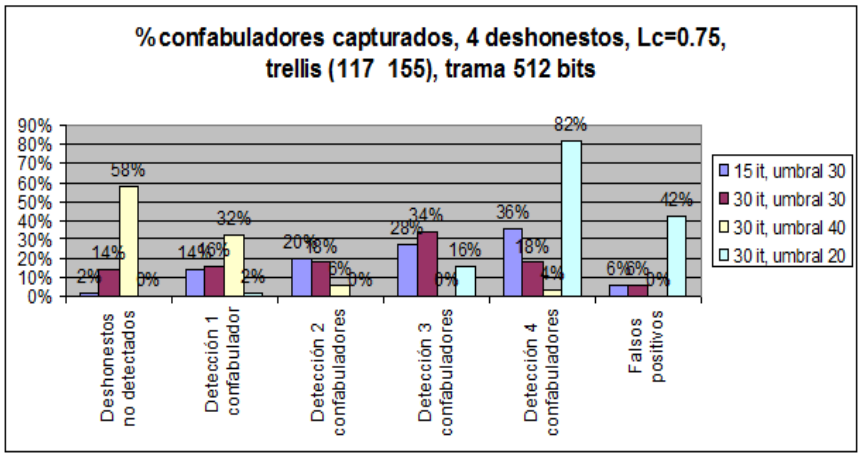


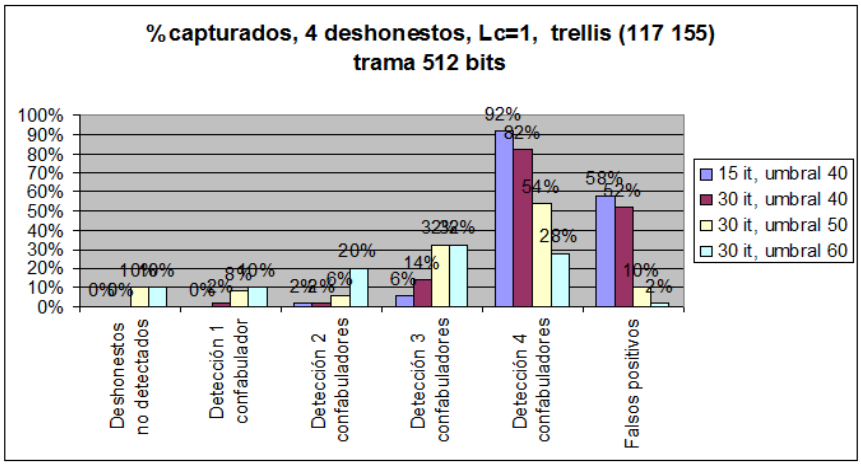
Figura 7.3: *% de captura de usuarios deshonestos generados con codificadores turbo de RSC* $(27, 31)_8$

Podemos observar en la figura 7.3 que al ampliar el valor de la constante de canal al valor Lc=1, nos aumenta de alguna manera la potencia de la señal decodificada y, en este caso, para un umbral de valor 40 tenemos un porcentaje de falsos positivos cercano 50 %. Por ello, pese a capturarse en la mayoría de los casos a todos los atacantes, no podemos asegurar para nada que la decodificación sea fiable. Se observa también que la decodificación es más uniforme, es decir, para el caso de colocar un umbral de 60, que es donde se tiene el límite para no detectar a ningún falso positivo, la distancia entre los porcentajes de no capturar ningún atacante y detectarlos a todos ellos se reduce. Este hecho puede ser debido a que ampliando el valor de la variable Lc estamos dando una mayor confianza a las secuencias que recibimos del canal, y estas secuencias están sumamente distorsionadas por los atacantes.

Aún así nos estamos poniendo en el caso de detectar 4 atacantes de entre mil, de modo que los valores no son excesivamente decepcionantes. De todos modos la situación de constante de canal de 0.75 y un umbral de 40 es la que más se adapta a los resultados esperados pues el porcentaje de obtener falsos positivos es cercano a 0, y tampoco se da la situación de no capturar a ningún atacante.



a)



b)

Figura 7.4: *Comparativa del rendimiento del filtro detector para RSC en el codificador turbo con trellis* $(117\ 155)_8$ *en a) Simulación realizada con Lc=0.75. b) Simulación realizada para Lc=1.*

La figura 7.4 muestra las mismas pruebas realizadas con trellis de mayor tamaño, $(117\ 155)_8$. Para poder comparar la mejora con el trellis $(27\ 31)_8$ se ha realizado el mismo número de iteraciones y se han utilizado los mismos niveles de umbral.

Para el caso de una constante de canal de 0.75, los resultados no son tan satisfactorios con este trellis mayor, ya que detectamos falsos positivos en todos los casos menos en el caso de utilizar un umbral de corte de 40, situación esta última que nos provoca que en el 58 % de las veces no detectemos atacante alguno. Por contra, en el caso de umbrales de corte de 30 observamos que las muestras son menos dispares, y a pesar de que se reduce notablemente el número de veces que obtenemos tan sólo un atacante, nos aumenta el número de veces que son los 4 que intervinieron en la confabulación los que son detectados.

En la figura 7.4.b utilizamos una constante Lc de 1. En ella se observa que los resultados son prácticamente idénticos que para el caso de un trellis menor. De todos modos, como aspectos a destacar, decir que aumenta ligeramente el porcentaje de falsos positivos y disminuye ligeramente el de deshonestos no detectados. No obstante, las mejoras son tan pequeñas que no podríamos decir que se ganase o perdiese rendimiento con el cambio de trellis para el caso de constante de canal de 1. Cabe destacar también que en el caso de utilizar un trellis mayor, $(117\ 155)_8$, el coste computacional se incrementa mucho y el tiempo que tarda en realizar las simulaciones se multiplica por tres.

Una vez realizadas las simulaciones con las distintas variables de la codificación turbo, se pasará a realizar una prueba sin codificación, para ver el resultado que se obtiene sin la aplicación de ningún método de codificación de canal. La siguiente figura muestra el resultado de lanzar tres procesos distintos de confabulación y decodificación por correlación realizados 50 veces cada uno con umbrales de decisión 150 y 160. Estos umbrales son mas elevados ya que los valores de las tramas son 0 y 1 porque no se ha realizado codificación.

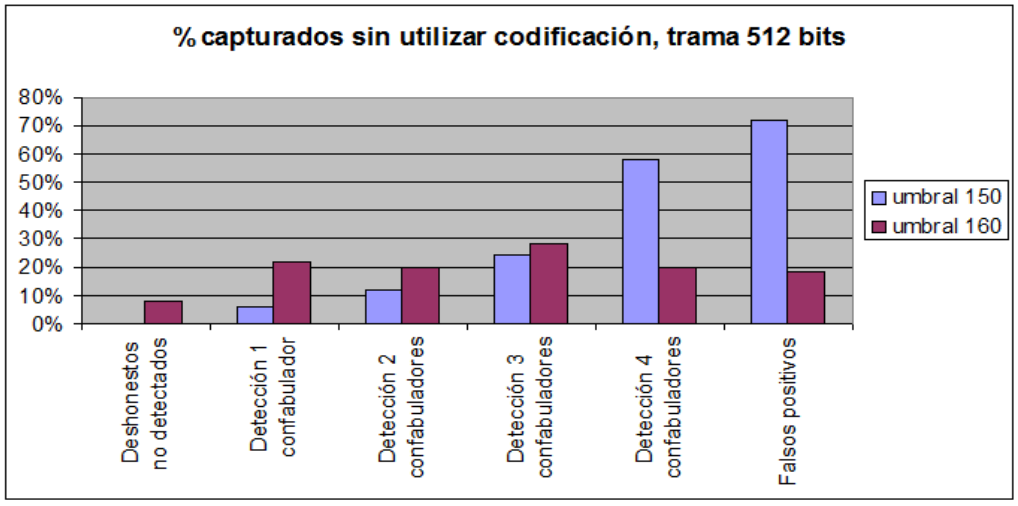


Figura 7.5: *% de captura de usuarios deshonestos sin realizar codificación de canal*

La gráfica 7.5 nos muestra unos resultados que debemos tener muy en cuenta, pues observamos que el hecho de no aplicar codificación a las tramas no significa que no seamos capaces de detectar los usuarios deshonestos por correlación, ya que para el caso de un umbral de 160 detectamos casi el 100 % de los usuarios y, aunque tengamos falsos positivos, los resultados son mejor de lo que cabía esperar. Estos resultados significan que la codificación turbo no es capaz de corregir suficientes bits erróneos como para introducir una mejora significativa que nos permita detectar a los usuarios confabuladores. Esta conducta es debida a que con tramas de una longitud de 512 bits tenemos 2^{512} tramas posibles y nuestro conjunto de estudio está compuesto solamente por 1000 usuarios. Este hecho conlleva que cada trama sea muy diferente entre sí, de modo que al obtener una trama formada por tríadas de bits entre 4 usuarios distintos, estos conservan el parecido entre sí, dicho de otro modo, la distancia de hamming entre los confabuladores es menor que la que tienen con el resto de usuarios una vez realizada la confabulación.

A continuación se muestran los resultados de una simulación con marcas digitales de menor longitud. Esta prueba se realiza para comprobar la eficacia de la codificación, ya que el hecho de utilizar tramas aleatorias de 512 bits puede comportar que las confabulaciones de 4 usuarios no enmascaren suficientemente la señal en un conjunto tan pequeño como 1000 usuarios. Es decir, con tan sólo una muestra de 1000 usuarios y 2^{512} tramas posibles, éstas tienen tantas diferencias entre sí que a pesar de que 4 usuarios con-

fabulen, es posible la detección por correlación. Este hecho no es deseable porque significaría que la capacidad del código no es un factor relevante en la detección de usuarios confabuladores. Por ello, la detección del código fallaría en el momento en que creciera la población de usuarios. Podemos hacer la comprobación de esta situación disminuyendo la longitud de la trama. A continuación se muestra una simulación realizada con tramas de 128 bits.

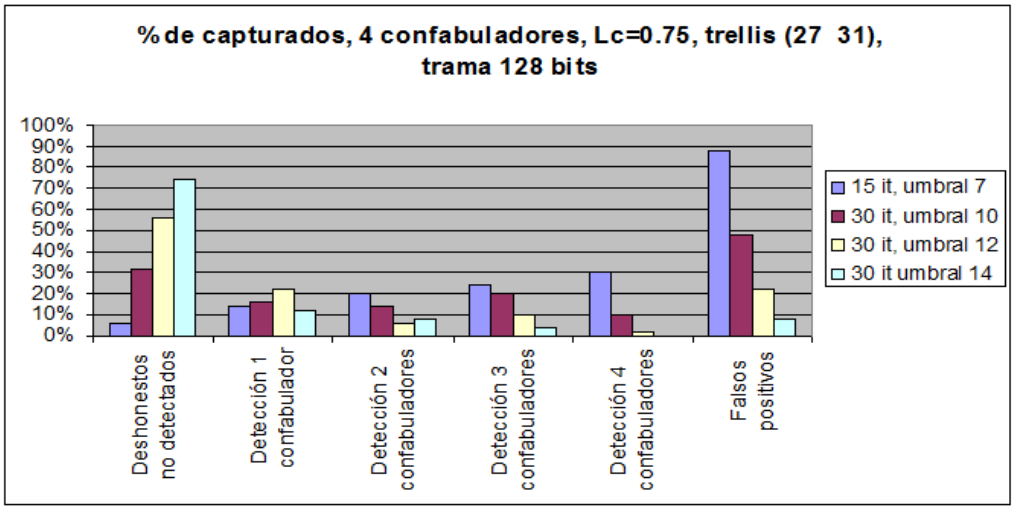


Figura 7.6: *% de captura de usuarios deshonestos utilizando tramas de 128 bits, Lc=0.75 y trellis (27 31)*

Como se puede ver en los resultados mostrados en la gráfica 7.5, la codificación y decodificación turbo falla para tramas pequeñas. Una de las características de la codificación turbo es que tiene un rendimiento mejor para secuencias largas que para secuencias cortas. De todos modos, el abrupto empeoramiento de la decodificación no es debido en este caso al rendimiento de la codificación sino al hecho de que, al haber menos tramas posibles cada una de ellas se parece más entre sí, por ello las correlaciones son mayores con los usuarios honestos. Este hecho resulta en la aparición de falsos positivos. Esto es debido a que la capacidad de corrección de los códigos turbo en las señales falsas se ve superada por parecidos fortuitos frente a señales no deshonestas.

Para comprobar la eficacia del código turbo realizaremos una decodificación con valores *hard*, obteniendo una marca digital nueva. Esta marca se compara bit a bit con el resto de usuarios, es decir medimos la distancia, también se compara la marca deshonesta con el resto de usuarios. La siguiente

figura muestra las simulaciones para los dos casos comentados.

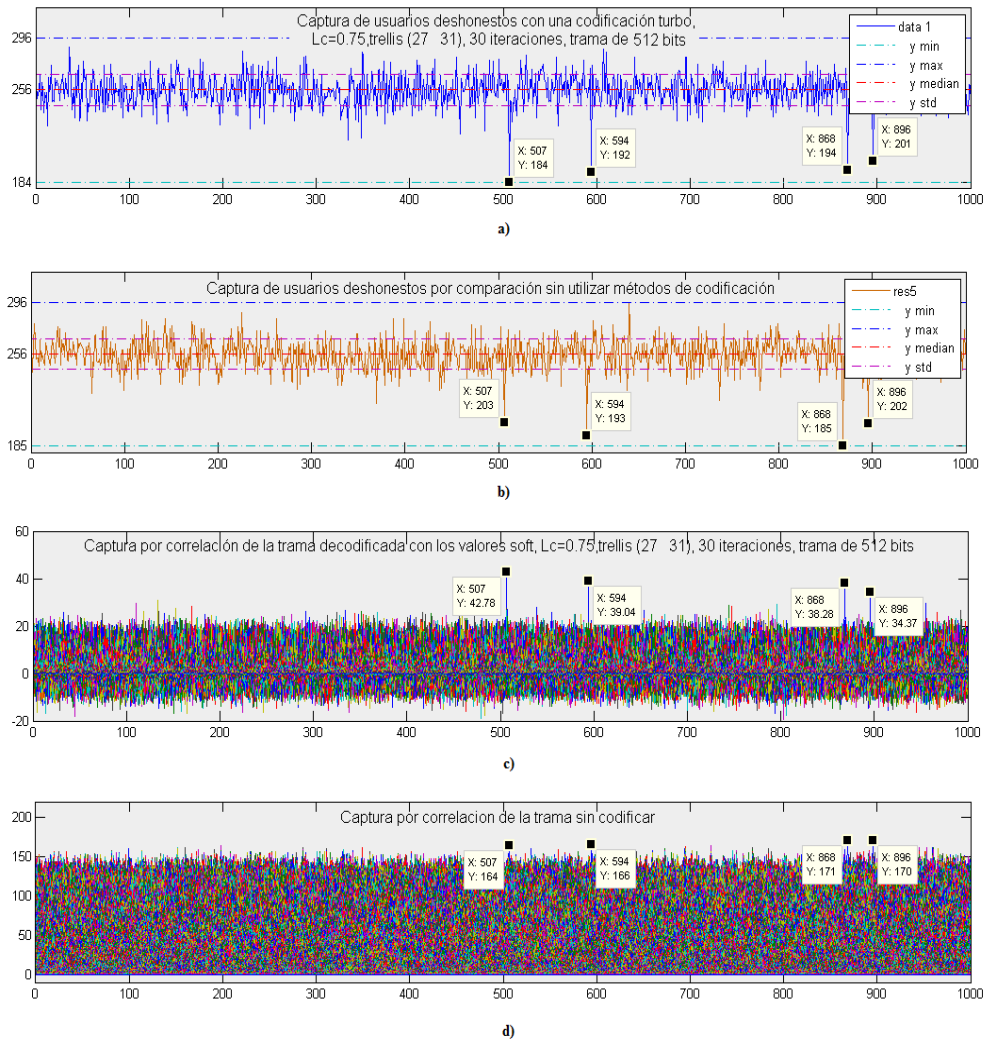


Figura 7.7: *comparación de las marcas falsas con el conjunto total de marcas. En a) la marca falsa ha pasado por un proceso de codificación turbo. En b) se comparan las marcas sin codificar. En c) se realiza una correlación de la marca con el resto de marcas del grupo. En d) se realiza la misma correlación pero con la marca sin ser codificada.*

La figura 7.7.a muestra la diferencia de bits existente entre la marca deshonesta y el resto de marcas del conjunto, mientras que la figura 7.7.b muestra la diferencia respecto a la marca deshonesta sin codificar. Como se

puede observar tanto para el caso de aplicar codificación turbo como para el caso de no aplicar ninguna codificación, es muy fácil encontrar a los usuarios deshonestos por simple comparación. Si nos fijamos en los cambios realizados por el codificador sobre la trama observamos que para el usuario 507 ha corregido 19 bits, pues esa es la diferencia sobre la trama original. No obstante, el usuario 868 ha empeorado 9 bits con esta corrección, y es que este es uno de los factores que añaden complejidad a la detección sobre ataques por confabulación utilizando técnicas de codificación de canal, porque el hecho de cambiar unos bits a favor de un usuario puede significar que añada más bits erróneos en otro. Por ello lo ideal sería que el codificador turbo tendiera a un usuario independientemente del resto, y que al menos en uno de ellos la corrección de errores fuera considerable. Pero este comportamiento, como podemos observar no se produce, de modo que lo único que logramos con la codificación es redistribuir los bits pero no recuperar una trama deshonesta con suficientes garantías.

Para el caso de la utilización de la correlación (figuras 7.7.c y 7.7.d) comprobamos que aquí si que hay mejora sobre el hecho de utilizar codificación turbo o no. Esto se debe a que estamos utilizando los valores $soft$ en el caso de que haya codificación, mientras que para el caso de que no la haya realizamos la correlación de unos y ceros. Este hecho corrobora los resultados obtenidos en la figura 7.5, donde se hallan falsos positivos juntamente con los usuarios deshonestos. De todos modos, el hecho que se puedan detectar sin problemas los usuarios deshonestos por comparación (figura 7.7.b) nos rebela que la codificación turbo no está siendo lo decisiva que cabría esperar.

Seguidamente mostraremos los resultados de la misma simulación pero con tramas mas cortas, 128 bits, para comprobar que en efecto el hecho de capturar a los usuarios deshonestos es debido a la probabilidad de las señales que no al efecto corrector del codificador turbo.

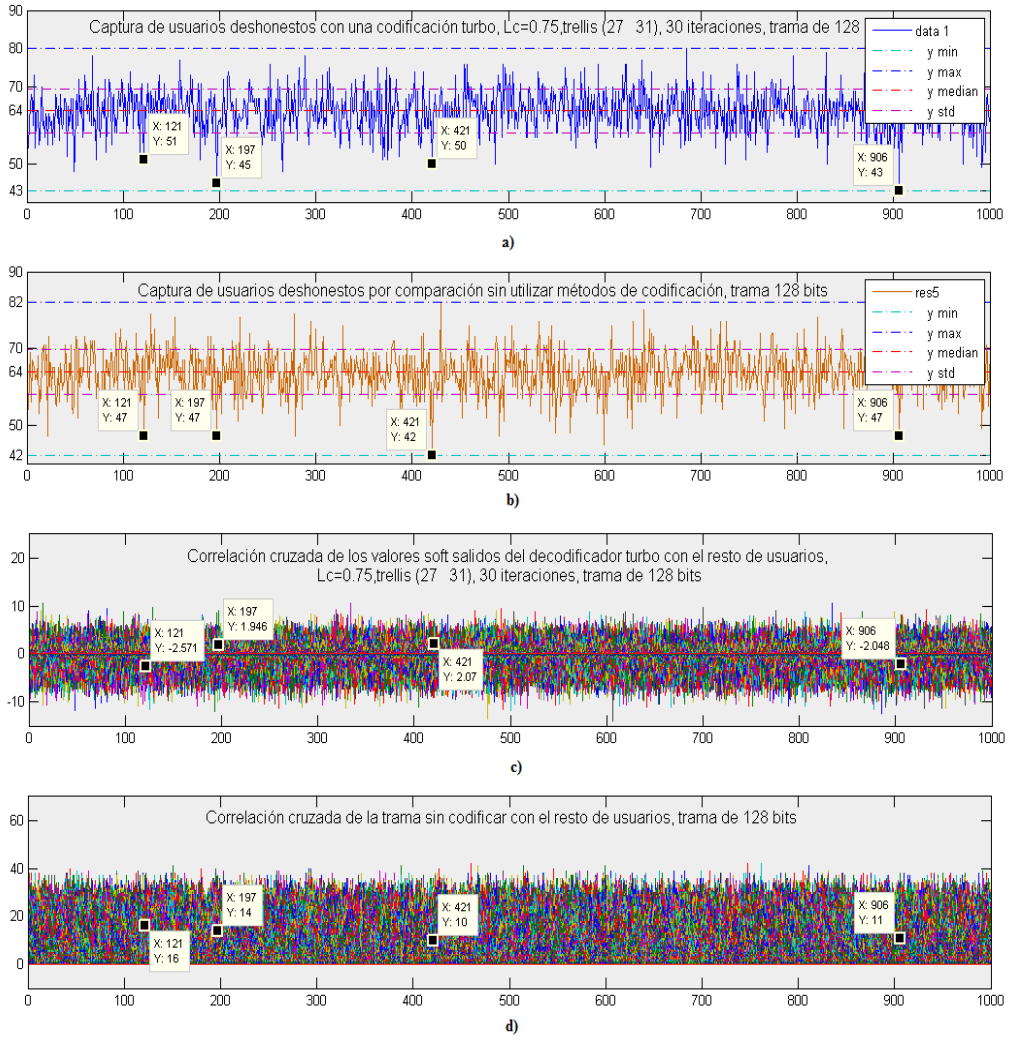


Figura 7.8: *comparación de las marcas falsas con el conjunto total de marcas. En a) la marca falsa ha pasado por un proceso de codificación turbo. En b) se comparan las marcas sin codificar. En c) se realiza una correlación de la marca con el resto de marcas del grupo. En d) se realiza la misma correlación pero con la marca sin ser codificada.*

Como se puede comprobar con un tamaño de marca de 128, la señal codificada casi no se distingue de la señal no codificada. En este caso los usuarios deshonestos son el 121, 197, 421 y 906. Como se puede ver en la figura 7.8.a, el mínimo lo marca el usuario 906. Aún así, hay muchos otros usuarios que por simple casualidad en la creación se parecen a la marca confabulada de forma

similar. Por otro lado si se compara la figura 7.8.a y 7.8.b, comprobamos que el mínimo también lo marca un usuario deshonesto. No obstante, en este caso es el usuario 421 con un valor de 42 bits distintos, de modo que para el caso de 128 bits también ha habido una distribución de los bits más que una corrección clara que nos reporte un usuario deshonesto con suficientes garantías. Así, con un tamaño de 128 bits por usuario, nos encontramos con muchas más dificultades para encontrar los usuarios deshonestos. De todos modos, de nuevo la mejora introducida por el codificador turbo es mínima, y en este caso comprobamos que dos usuarios, el 121 y el 421, tienen menos parecido con la marca falsa después de pasarla por un proceso de codificación de canal.

Para el caso de utilizar correlación como método de detección (figuras 7.8.a y 7.8.b), con usuarios de 128 bits ya no somos capaces de detectar con garantías a ningún usuario. Como se ha apuntado anteriormente los motivos por lo que los resultados empeoran tan radicalmente se debe mayoritariamente a que las marcas son mucho mas parecidas entre sí ahora que antes ya que el conjunto de marcas posibles se ha reducido por 4. En menor medida, creemos que la degradación también se debe a que los códigos turbo pierden rendimiento con señales mas cortas. Por último, se puede deber al diseño del interleaver utilizado, ya que en el caso de 128 bits el interleaver se ha generado a través de la función de Matlab **sort(rand(1,128))**, mientras que los interleavers de 512 bits utilizados por los codificadores convolucionales se han extraído de la siguiente página web:
http://www.es.lth.se/home/jht/interleaverdownload.html.

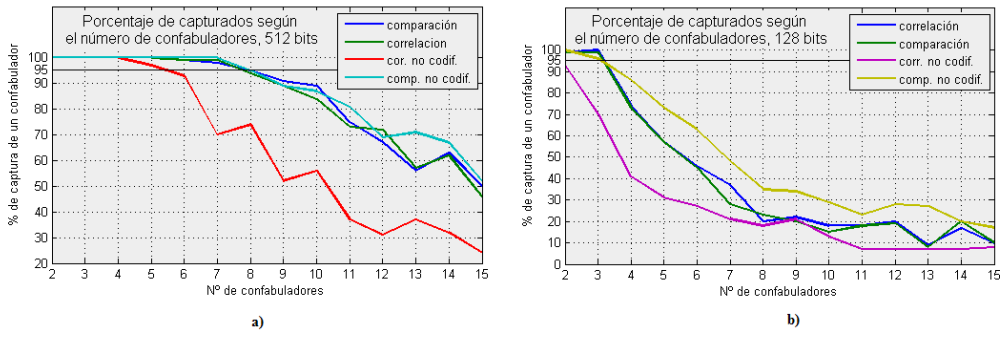


Figura 7.9: *% de captura de un deshonesto por comparación y correlación para distinto número de usuarios. En a) para tramas de 512 bits y en b) para 128 bits.*

Las figuras 7.9.a y 7.9.b muestran la evolución en la capacidad de cap-

tura de un usuario deshonesto para el caso de usuarios con 512 y 128 bits. En este caso el criterio ha sido quedarnos con el usuario que presenta un mínimo en caso de utilizar comparación, o bien un máximo en caso de utilizar correlación. Los resultados obtenidos reflejan que, para el caso de utilizar 512 bits, se detecta a un usuario confabulador en prácticamente el 100 % de los casos hasta 7 usuarios deshonestos para todos los métodos de captura a excepción de la correlación sin utilizar codificación turbo. A partir de los 7 usuarios, el rendimiento va empeorando paulatinamente hasta no encontrar a ningún usuario en más de la mitad de las ocasiones para más de 15 usuarios confabuladores. Hay que observar que el simple hecho de comparar las marcas sin utilizar confabulación nos reporta un rendimiento igual o incluso mejor que realizar una codificación turbo a las tramas. Este hecho se debe, como se ha dicho anteriormente, a que el codificador no es capaz de aislar a un usuario en su decodificación porque la capacidad de corrección se ha superado debido tanto a la naturaleza del error como a la cantidad de bits modificados. De esta manera, es posible que el codificador resuelva correctamente qué bit debe ir en una posición en concreto, más no es capaz de realizar esta corrección para un usuario en concreto y de forma suficiente. Por contra el codificador realiza pequeños cambios que en unas ocasiones nos acercan más a un usuario pero nos alejan más del parecido con otros, y es por este motivo que en muchas ocasiones el hecho de utilizar codificación de canal empeora el comportamiento.

Para el caso de usuarios de 128 bits, (figura 7.9.b) observamos que la caída en el rendimiento es mucho más acentuada y que para el caso de más de 3 confabuladores el porcentaje de capturas de un usuario deshonesto ya está por debajo del 95 %, este umbral se sobrepasa a partir de los 7 usuarios en caso de utilizar tramas de 512 bits. Para el caso de más de 15 usuarios deshonestos el porcentaje de capturas es del orden del 10 %, capturas que se deben más a factores estadísticos que a la propia capacidad del código turbo. Este hecho corrobora que el factor determinante para las detecciones de usuarios es la longitud de la trama, es decir, la capacidad de crear tramas muy distanciadas entre sí, más que la capacidad del código corrector.

7.2. Confabulación por comparación bit a bit

Analizaremos en esta parte el comportamiento del esquema de codificación-decodificación turbo para el caso de que los atacantes comparen las secuencias

a nivel de bit y se queden con un bit de cada trama aleatoriamente. Para comparar la eficacia de este tipo de ataque por confabulación se realizarán las mismas pruebas que en el caso de la confabulación mediante palabras de tres bits.

7.2.1. Detección de dos confabuladores

Como en el caso anterior, aquí lo que se pretende es comparar dos marcas válidas para obtener una marca falsa mediante la elección aleatoria de los bits en las posiciones de las tramas en que éstos difieran.

La siguiente figura muestra el resultado obtenido de la simulación del proceso de creación de marca y confabulación realizado un total de 50 veces.

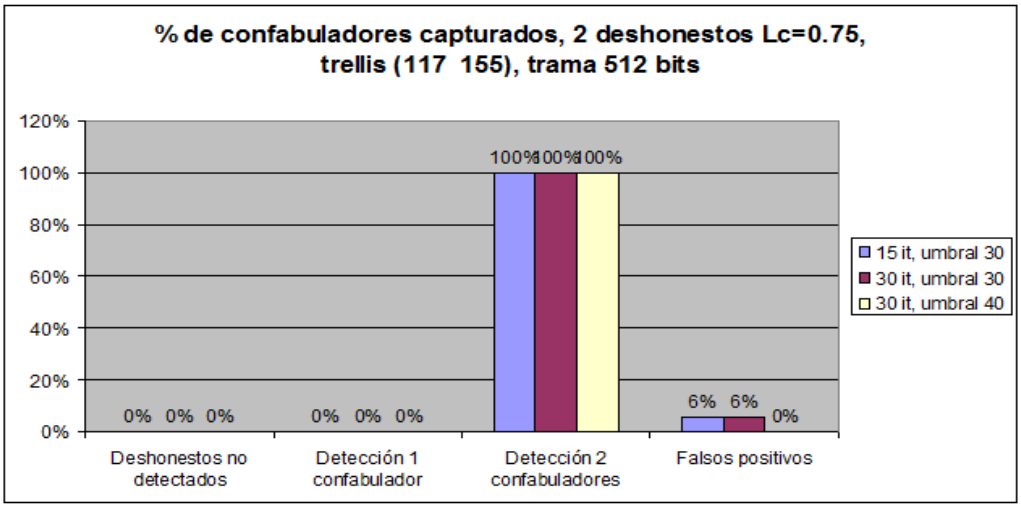
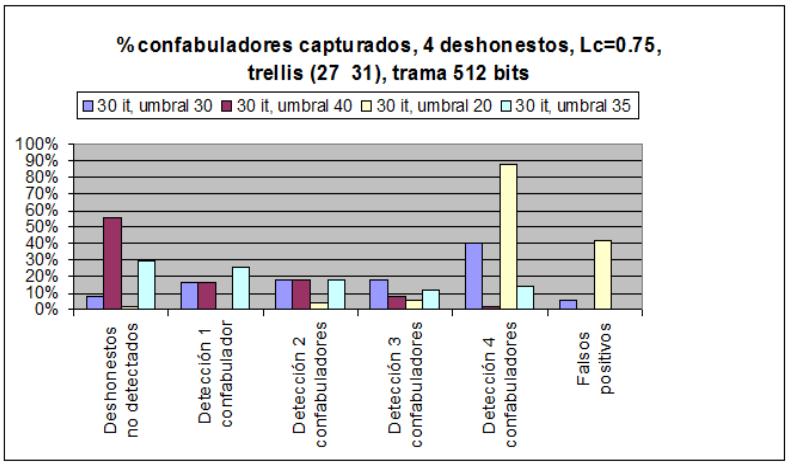


Figura 7.10: *% de captura de usuarios deshonestos generados con codificadores turbo de RSC* $(27, 31)_8$

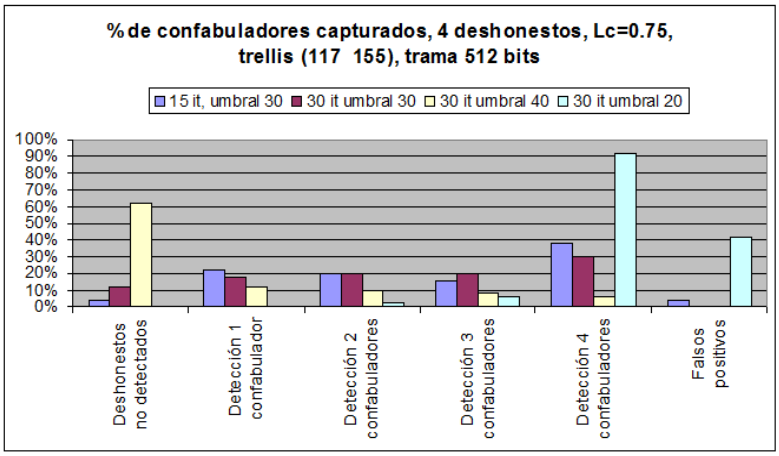
Como podemos comprobar, no hay problema para obtener a los confabuladores que han participado en la obtención de la marca. En todos los casos se recuperan las marcas originales, aunque en un principio se cuelan usuarios honestos al utilizar un umbral de corte de 30. Al subir este valor a 40 los resultados son satisfactorios y no obtenemos ningún error en detección.

7.2.2. Detección de 4 confabuladores

La siguiente gráfica nos muestra los resultados obtenidos en la simulación de 4 usuarios confabulando mediante la comparación bit a bit, con una constante de canal de 0.75 y entrelazadores para los codificadores constituyentes de los codificadores turbo de (27 31) y (117 155).



a)



b)

Figura 7.11: *% de captura de usuarios deshonestos generados con codificadores turbo RSC* $(27\ 31)_8$ *en a) y RSC* $(117\ 155)_8$ *en b). Constante de canal Lc=0.75. Comparación bit a bit.*

En esta situación, la distribución de los resultados es ligeramente distinta que para el caso de confabulación en bloques de tres bits. Si comparamos los valores de la figuras 7.2 y 7.8, observamos que, para el último caso, el número de deshonestos no detectados para un umbral de 40 y 30 iteraciones es mayor de el 50 %, mientras que para el caso anterior se capturaban el 100 % de los atacantes. Por otro lado, el número de veces en que se capturan los 4 atacantes es mayor para este último estudio de una confabulación bit a bit, donde el resultado parece ser un poco más homogéneo que para el estudio de ataque por bloques, aunque el rendimiento en la corrección es menos acentuado.

Esta situación era de esperar, puesto que al confabular a nivel de bit la degradación de la marca es más homogénea y la estadística de degradación más distribuida. Esto significa que, en comparación con el caso de confabulación a nivel de palabra, si aleatoriamente nos quedamos con las palabras del mismo usuario tres veces seguidas, significa una secuencia de 9 bits seguida de la misma marca, mientras que para el caso de los bits tan sólo nos quedamos con tres bits. En el caso de confabulación por palabras recordemos que en un porcentaje muy elevado capturábamos un confabulador debido seguramente a este factor que comentamos de la aleatoriedad en la elección de palabras de una u otra marca.

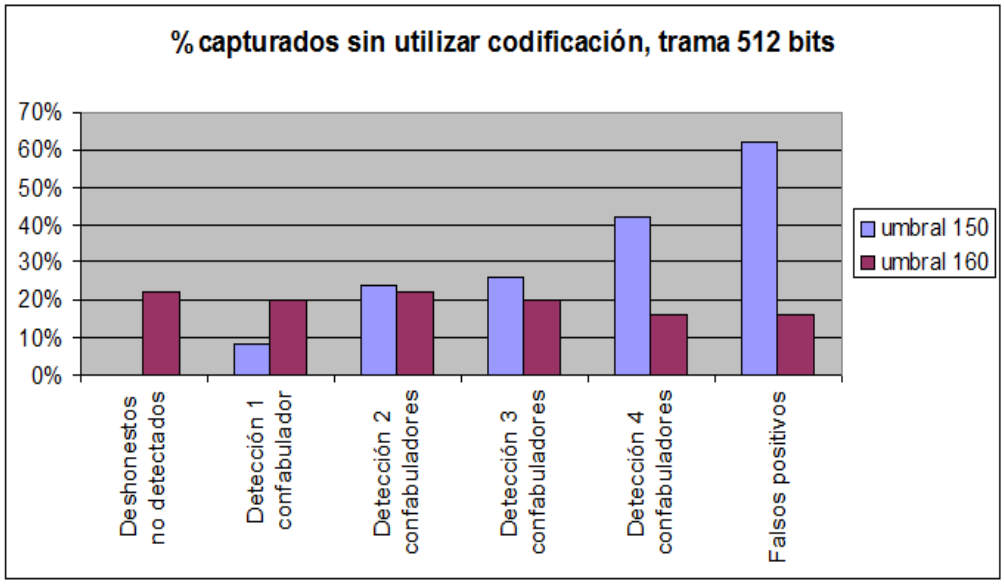


Figura 7.12: *% de captura de usuarios deshonestos sin utilizar codificación de canal*

Como observamos en la figura anterior, sin utilizar codificación de canal los resultados son menos fiables en el sentido que los usuarios quedan enmascarados con el resto de usuarios honestos del grupo. Este comportamiento es el que se refleja en las gráficas 7.7 y 7.14, las cuales muestran que la correlación *hard* de las tramas sin codificar presenta un rendimiento peor que la correlación de los valores *hard*. De la misma manera que en el caso anterior el hecho de poder capturar a los confabuladores es debido a que los 512 bits de trama son muy grandes respecto al número de confabuladores presentes.

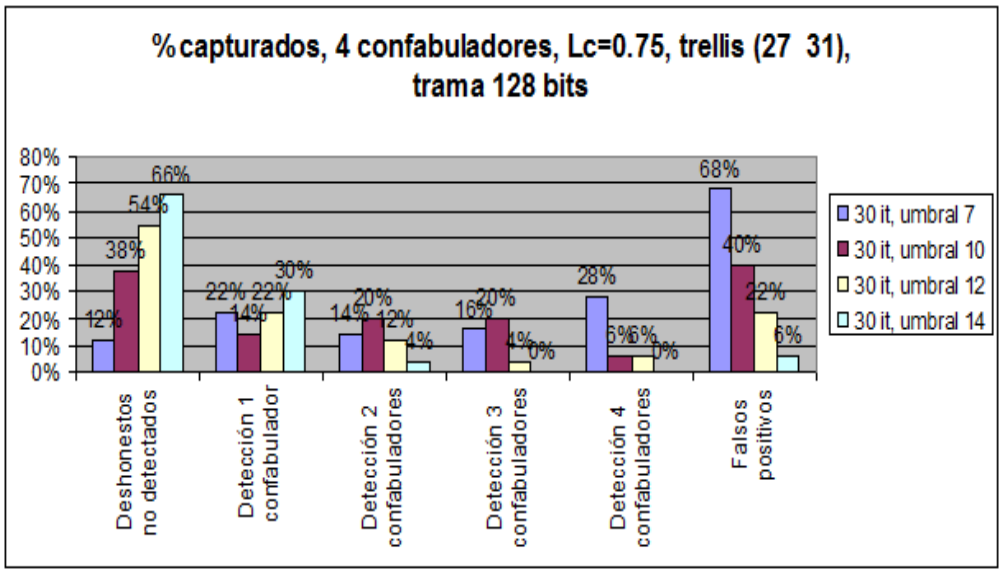


Figura 7.13: *% de captura de usuarios deshonestos utilizando tramas de 128 bits, Lc=0.75 y trellis(25 31). Confabulacion por comparación bit a bit.*

Al igual que pasaba con el caso de una confabulación por bloques de tres bits, ahora los resultados son prácticamente idénticos, es decir, con una longitud de marca de 128 bits los resultados empeoran muy significativamente respecto a los resultados con una longitud de trama de 512. Las causas, en este caso, son las mismas que para el caso anterior. Igual que hicimos en las figuras 7.7 y 7.8, seguidamente mostraremos el rendimiento de la codificación turbo mediante la comparativa de la marca deshonesta confabulada con el resto de marcas del conjunto. La prueba se realizará para longitudes de trama de 512 y 128 bits.

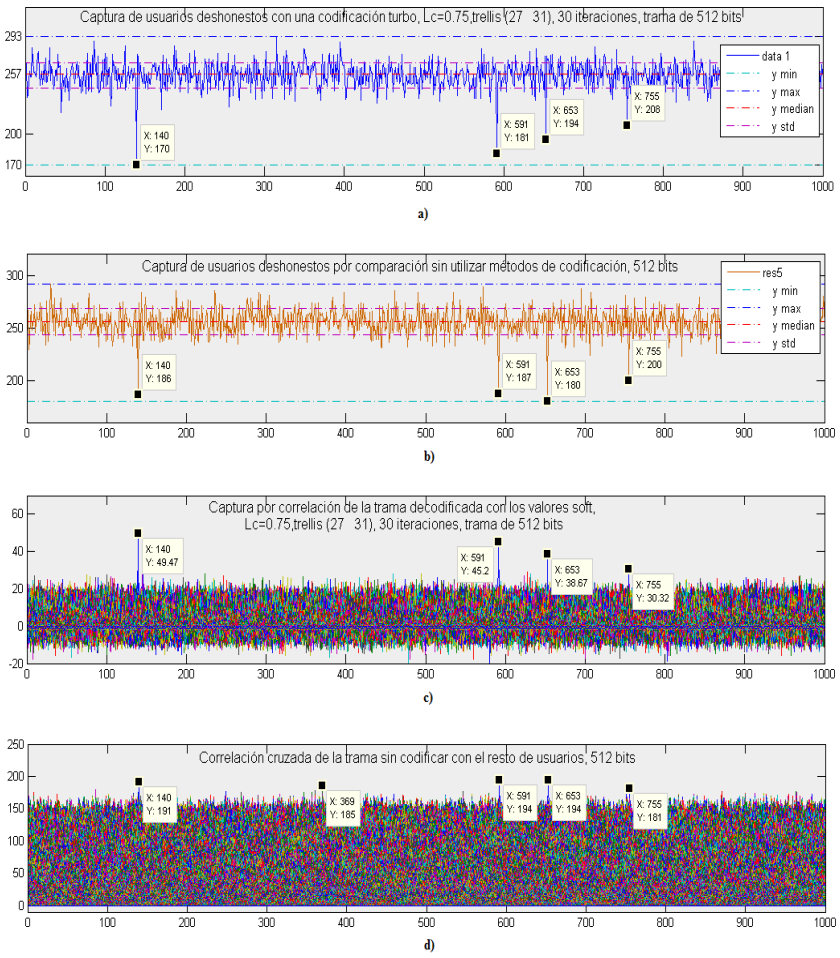


Figura 7.14: *comparación de las marcas falsas con el conjunto total de marcas. En a) la marca falsa ha pasado por un proceso de codificación turbo. En b) se comparan las marcas sin codificar. En c) se realiza una correlación de la marca con el resto de marcas del grupo. En d) se realiza la misma correlación pero con la marca sin ser codificada. Confabulacion por comparación bit a bit.*

Se corrobora, en los resultados mostrados en la figura 7.14, que con una longitud de trama de 512 bits no hay problema para detectar los confabuladores. Este mismo comportamiento es el mismo que el observado en la figura 7.7, por lo que podemos decir que la naturaleza de la confabulación no es un factor determinante si el tamaño de la trama es suficientemente grande. Otra vez se puede comprobar el efecto del codificador turbo, pues el usuario 140 pasa de una diferencia de 186 bits respecto a la marca confabulada a una

diferencia de 170 bits. Esto es, al menos se han cambiado 16 posiciones que han hecho que la marca se acerque más a éste usuario pero se distancie, por ejemplo, del usuario 653. En el caso de utilizar correlaciones observamos que el hecho de no utilizar codificación turbo nos da un resultado más pobre, y se cuela el usuario 369 como falso positivo. Este hecho, como se ha comentado, es debido a que no se utilizan valores *soft* como en el caso de una trama turbodecodificada. Observaremos en la figura siguiente cuál es el resultado para una longitud de trama de 128 bits.

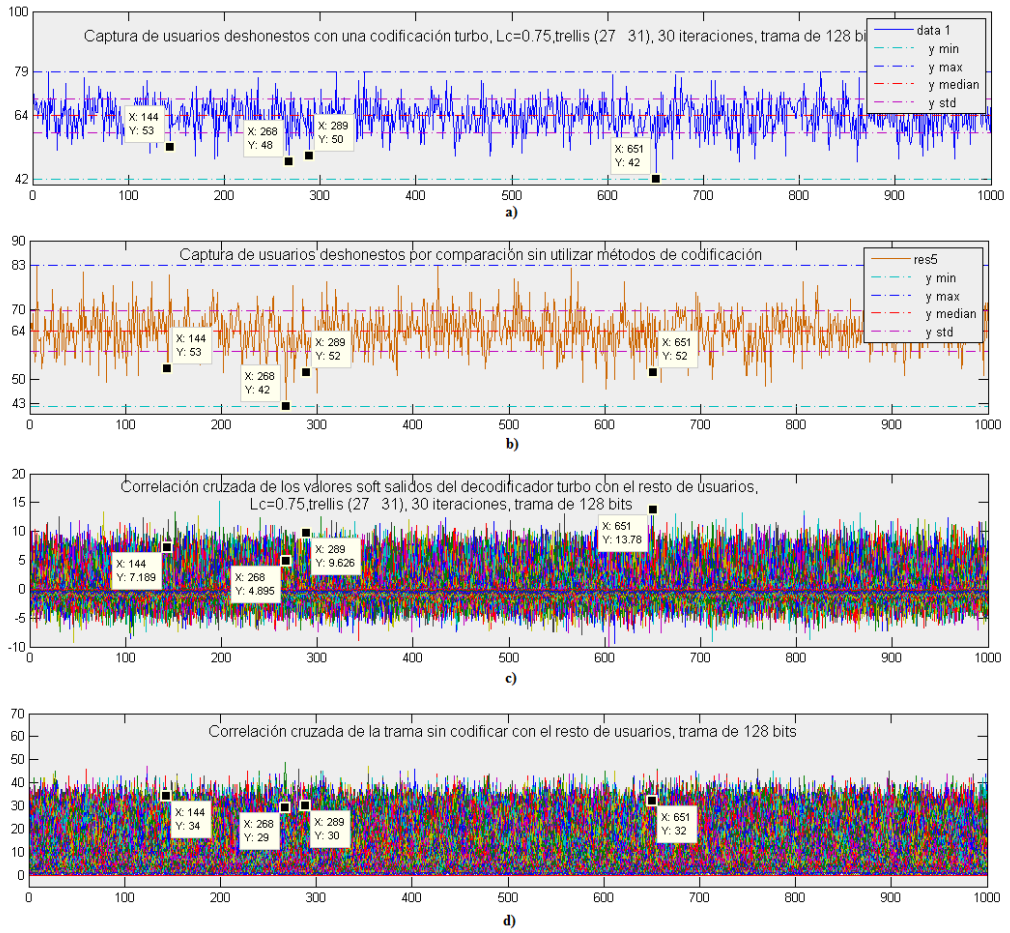


Figura 7.15: *a) comparación de marca que ha pasado por un proceso de codificacion turbo. En b) se comparan la marca sin codificar. Confabulacion por comparación bit a bit. trama de 128 bits*

En la gráfica 7.15 se observa que el rendimiento es menor que en el caso

de 512 bits, del mismo modo que pasaba en el caso de la confabulación por bloques mostrado en la figura 7.8, donde el mínimo estaba en una diferencia de 43 bits. En este último caso, el mínimo está en 42 bits para el usuario codificado y 42 bits para el que no se ha aplicado codificación. Otra vez se demuestra que la correlación con tramas cortas no tiene un resultado satisfactorio. Esto demuestra que utilizar tramas pequeñas conlleva usuarios muy parecidos entre sí, y por ende menos capacidad de detección de confabuladores, con codificación o sin ella.

Del mismo modo que hicimos en el caso de bloques de tres bits en la figura 7.9, la figura siguiente muestra el rendimiento en la detección de un usuario a medida que aumentamos el número de usuarios confabuladores.

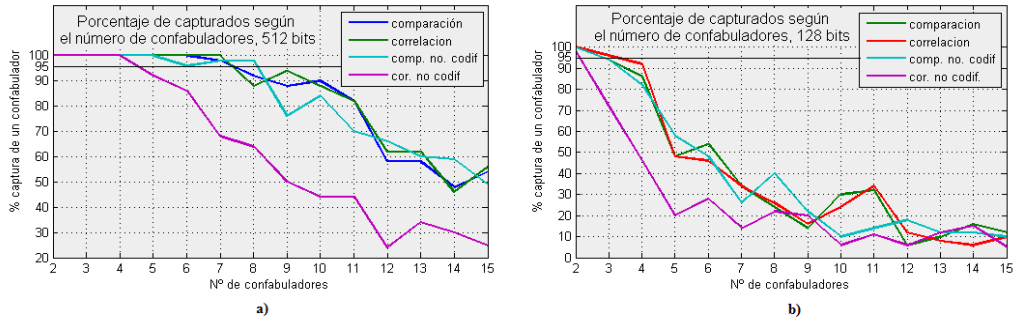


Figura 7.16: *% de captura de un deshonesto por comparación y correlación para distinto número de usuarios. En a) para tramas de 512 bits y en b) para 128 bits.*

Del mismo modo que ocurría para una confabulación por bloques de tres bits, nos encontramos, en esta ocasión, que el hecho de utilizar codificación turbo no es determinante para encontrar a los confabuladores. Tal y como se puede observar en la figura 7.16, el éxito en las capturas lo marca más bien el tamaño de la trama de usuario antes que el hecho de utilizar codificación o no. Así, en la figura 7.16, como ocurría en la figura 7.9, la captura por comparación de usuarios sin utilizar codificación tiene un rendimiento similar al obtenido utilizando codificación turbo, y tan sólo la correlación nos reporta un rendimiento peor.

Capítulo 8

Conclusiones y líneas futuras de investigación

8.1. Conclusiones

A la vista de los resultados obtenidos en las simulaciones realizadas en este proyecto, debemos concluir que el uso de la codificación turbo para la creación de *fingerprints* que protejan el copyright de una imagen, video o cualquier otro medio digitales es una solución que debe ser estudiada en más profundidad para solucionar las debilidades que los resultados de este proyecto reflejan en la captura de usuarios deshonestos, que han creado un usuario falso mediante un ataque por confabulación. Los resultados expuestos en las figuras 7.7, 7.9.a, 7.14 y 7.16.a nos rebelan que la captura de usuarios deshonestos ha sido debida al hecho que en un conjunto de muchos usuarios posibles, 2^{512} en nuestro caso, la distancia de hamming entre un grupo pequeño de usuarios, 1000 en nuestro caso de estudio, es lo suficientemente grande como para que un ataque por confabulación no aumente lo suficiente la distancia de hamming entre la marca falsa y los respectivos confabuladores. Este hecho conlleva que por simple comparación, localicemos los usuarios atacantes. Por otro lado, con un número mayor de usuarios o bien un conjunto de usuarios mucho menor, en nuestro caso 2^{128}, la distancia de hamming entre usuarios es mucho menor. Por ello, la capacidad de detección disminuye considerablemente. Este comportamiento se puede observar en las figuras 7.8, 7.9.b, 7.15 y 7.16.b, en las cuales nos damos cuenta que la codificación no mejora la

capacidad de detección de confabuladores más allá de la correlación, hecho debido más a la naturaleza de la señal que a la propia capacidad del código turbo.

Las causas de este comportamiento pueden ser las siguientes:

- **La naturaleza del error:** recordemos que estamos aplicando codificación de canal para dar robusteza a unas marcas que van a ser modificadas no por la distorsión de un canal de comunicación, sino por un ataque por comparación realizado intencionadamente una vez detectadas la longitud y posición de la señal dentro del medio marcado. De este modo, si recuperamos la naturaleza de la codificación turbo utilizada (ecuaciones 5.1-5.15), todas las métricas utilizadas en la decodificación turbo mediante el algoritmo BCJR se han hecho basándose en una estadística AWGN debido a un canal de transmisión Gausiano y sin memoria. Estas ecuaciones son las que se han utilizado para el diseño de la plataforma de codificación turbo de este proyecto, por ello es posible que la capacidad correctora no tenga el rendimiento esperado.
- **La naturaleza de la señal:** en este caso también tenemos un factor relevante en la señal que le llega al decodificador turbo. Como se ha dicho, el diseño del codificador se ha creado para que a la entrada le llegue una señal con una estructura de una señal codificada mediante un codificador turbo de ratio $\frac{1}{3}+$ ruido AWGN, y lo que en realidad recibe es una señal compuesta por fracciones de señales codificadas. Este hecho resulta no solamente en la degradación de los bits recibidos, si no en el hecho que en el mejor de los casos el decodificador debería realizar una decisión en decodificación, decisión que no ha sido contemplada. Recordemos que, por ejemplo, en una confabulación de 4 usuarios tenemos 4 marcas turbocodificadas en una sola, de modo que *a priori* ya estaba contemplado que el decodificador no fuera capaz de devolvernos un usuario original, descartando los otro tres. El comportamiento esperado fuera que la codificación realizara suficientes correcciones sobre un usuario para que este quedara delatado. Como se desprende de las simulaciones, hay correcciones de bits, pero éstas no son suficientes ni están localizadas en un usuario en concreto. Por ello, lo que se produce es una distribución de los bits sobre la señal, que aumenta la distancia de hamming sobre unos confabuladores del mismo modo que la reduce sobre otros. Por ello la ganancia de la codificación es tan pobre.
- **La naturaleza de las marcas:** no debemos perder de vista que una de

las situaciones no deseadas en la creación de las marcas es el hecho que sin realizar codificación podamos recuperar los usuarios casi del mismo modo que con una codificación turbo. Por ello, se puede pensar que la creación de los usuarios de modo aleatorio no es óptimo. Una buena línea de estudio en este ámbito sería la creación de marcas con una estructura diseñada para sacar el máximo provecho de la codificación de canal frente a ataques por confabulación aumentando, de este modo, el rendimiento sobre la detección de usuarios sin utilizar codificación alguna.

8.2. Líneas futuras de investigación

A continuación se presentan algunas posibles líneas de investigación relacionadas con la temática de este estudio.

En primer lugar y a la vista del rendimiento de las marcas de usuario sin utilizar codificación de canal, creemos que una buena línea de investigacón podría ser el estudio de estructuras de marca que fueran más robustas a los ataques de confabulación,ya sea por si solas o diseñadas pensando en una posterior codificación turbo del usuario. Esta línea de investigación debería estar encaminada a disminuir la distancia de hamming de una marca falsa obtenida por métodos de confabulación, de manera que esta marca deshonesta fuese transparente a detecciones por comparación sin utilizar métodos de codificación de canal. Así mismo, las propiedades de la marca deberían hacer posible que las mejoras obtenidas una vez se haya aplicado una codificación de canal a éstas fuesen considerables; así, ya sea por medio de los valores $soft$ del decodificador o por medio de la completa decodificación de la señal, los resultados de la codificación deberían reportar al menos un usuario deshonesto con suficiente claridad.

La segunda línea de investigación que se propone es la de la combinación de codificación bloque como puede ser los códigos Reed-Solomon o los BCH juntamente con los códigos turbo para dar mayor robustez a las secuencias. Esto, que ya se realiza para comunicaciones espaciales podría ser útil para resolver ataques por confabulación. Creemos que codificaciones basadas en métodos algebraicos pueden añadir robustez frente a la combinación de múltiples palabras turbo codificadas. Estos métodos algebraicos podrían ayudar a desglosar las diferentes palabras código turbo, que en cierta manera, forman

parte de la marca falsa, ayudando de este modo a la posterior decodificación turbo.

Por último, cabe incluir la posibilidad que la codificación de canal diseñada para errores producidos por un canal de transmisión no sea válido para la corrección de errores producidos por ataques intencionados por confabulación. Teniendo en cuenta esto, otra linea de estudio sería la creación de códigos específicos para esta problemática, siempre teniendo en cuenta que la captura final de usuarios deshonestos se hará por comparación en una muestra relativamente pequeña de usuarios. Así, es posible el diseño de estructuras algebraicas específicas para conjuntos pequeños de usuarios que, por ejemplo delaten la codificación por bloques o a nivel de bit de usuarios dentro de esta muestra.

Apéndice A

Polinomios y campos binarios

Un polinomio $f(X)$ con una variable X y con coeficientes de un campo $GF(2)$ tiene la forma

$$f(X) = f_0 + f_1 X + f_2 X^2 + \cdots + f_n X^n \tag{A.1}$$

donde el grado del polinomio es la potencia más alta de X con un coeficiente distinto a cero.

Podemos construir códigos con símbolos de cualquier campo de Galois $GF(q)$. No obstante, nos centraremos en los más utilizados, binarios $GF(2)$ y sus extensiones $GF(2^m)$.

Los polinomios bajo $GF(2)$ pueden ser sumados, multiplicados y divididos de la forma habitual, sean $g(X)$ polinomio de grado m y $f(X)$ polinomio de grado n. Si asumimos que $n > m$, entonces:

$$\begin{aligned} f(X) + g(X) &= (f_0 + g_0) + (f_1 + g_1)X + \ldots \\ &+ (f_m + g_m)X^m + fm + 1X^{m+1} + \cdots + f_n X^n \end{aligned} \tag{A.2}$$

$$f(X) \cdot g(X) = c_0 + c_1 X + c_2 X^2 + \cdots + c_{n+m} X^{n+m} \tag{A.3}$$

donde:

$$\begin{aligned} c_0 &= f_0 g_0 \\ c_1 &= f_0 g_1 + f_1 g_0 \\ &\vdots \\ c_i &= f_0 g_i + f_1 g_{i-1} + f_2 g_{i-2} + \cdots + f_i g_0 \\ &\vdots \\ c_{n+m} &= f_n g_m \end{aligned} \tag{A.4}$$

Los polinomios bajo GF(2) cumplen las siguientes propiedades:

- Conmutativa:

$$a(X) + b(X) = b(X) + a(X) a(X) \cdot b(X) = b(X) \cdot a(X) \tag{A.5}$$

- Asociativa:

$$\begin{aligned} a(X) + [b(X) + c(X)] &= [a(X) + b(X)] + c(X) \\ a(X) \cdot [b(X) \cdot c(X)] &= [a(X) \cdot b(X)] \cdot c(X) \end{aligned} \tag{A.6}$$

- Distributiva:

$$a(X) \cdot [b(X) + c(X)] = [a(X) \cdot b(X)] + [a(X) \cdot c(X)] \tag{A.7}$$

Para la división se cumple la siguiente ecuación:

$$f(X) = q(X) g(X) + r(X) \tag{A.8}$$

Esto es, al dividir $f(X)$ entre $g(X)$ obtenemos un único par de polinomios $q(X)$ y $r(X)$ que son cociente y residuo respectivamente. El grado de $r(X)$ es menor que $g(X)$. Esta división se llama *algoritmo de la división Euclidea.* Veamos un ejemplo:

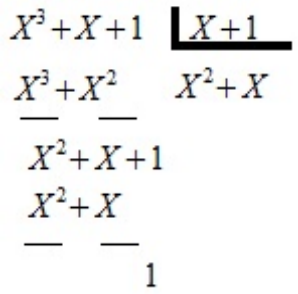


Figura A.1: *division polinómica*

Como vemos en este ejemplo nos queda un residuo de $r(X) = 1$ y $q(X) = X^2 + X$. Sería fácil demostrar la siguiente ecuación:

$$X^3 + X + 1 = (X + 1)(X^2 + X) + 1 \tag{A.9}$$

Definición 2 *Un elemento del campo a es cero o raíz del polinomio $f(X)$ si $f(a) = 0$. En este caso se dice que a es raíz de $f(X)$ y, entonces, $X - a$ es factor del polinomio $f(X)$.*

Definición 3 *Se dice que un polinomio $p(X)$ bajo $GF(2)$ de grado m es irreducible bajo $GF(2)$ si $p(X)$ no es divisible por ningún otro polinomio $GF(2)$ de grado menor a m pero mayor de 0.*

Teorema 4 *Cualquier polinomio irreducible en GF(2) de grado m divide a $X^{2^{m-1}} + 1$.*

Se dice que un polinomio irreducible $p(X)$ de grado m es primitivo si el menor entero n por el cual $p(x)$ divide a $X^n + 1$ es $n = 2^m - 1$. Para un m dado, puede haber más de un polinomio primitivo de grado m. No es fácil encontrar estos polinomios; por ello, hay tablas donde aparecen listas de polinomios primitivos. Otra propiedad interesante de un polinomio bajo $GF(2)$ es la siguiente:

$$(f(X))^{2^l} = f(X^{2^l}) \tag{A.10}$$

A.1. Construcción de un Campo de Galois $GF(2^m)$

Un campo extendido de Galois $GF(2^m)$ contiene como elementos, además del 0 y el 1, el elemento α y sus potencias. Se cumple lo siguiente:

$$\begin{aligned} &0 \cdot \alpha = \alpha \cdot 0 = 0 \\ &1 \cdot \alpha = \alpha \cdot 1 = \alpha \\ &\alpha^2 = \alpha \cdot \alpha, \alpha^3 = \alpha^2 \cdot \alpha \\ &\alpha^i \cdot \alpha^j = \alpha^{i+j} = \alpha^j \cdot \alpha^i \end{aligned} \tag{A.11}$$

La multiplicación queda definida dentro de un conjunto de elementos F siguiente:

$$\mathbf{F} = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^j, \ldots\} \tag{A.12}$$

Seguidamente, definimos la condición que el conjunto F contiene tan solo $2m$ elementos y que es cerrado respecto a la multiplicación.

Sea $p(X)$ un polinomio primitivo de grado m bajo $GF(2)$. Asumimos, también que α es raíz de $p(X)$. Esto es, $p(\alpha) = 0$. Al ser $p(X)$ primitivo, tenemos que divide a $X^{2^m-1} + 1$:

$$X^{2^m-1} + 1 = q(X)p(X) \tag{A.13}$$

Al ser α raíz de $p(X)$, obtenemos:

$$\begin{aligned} &\alpha^{2^m-1} + 1 = q(\alpha) \cdot 0 \\ &\alpha^{2^m-1} + 1 = 0 \\ &\alpha^{2^m-1} = 1 \end{aligned} \tag{A.14}$$

Estos elementos distintos de zero del conjunto F forman un grupo conmutativo bajo la multiplicación " $\cdot$ " . Además, se puede demostrar que $\alpha^{2^m-1} + 1$ es el multiplicando inverso de α^i.

Así tenemos que el conjunto F es un grupo de orden $2^m - 1$ con respecto a la multiplicación. El siguiente paso es asegurarnos que la operación de suma es conmutativa dentro de F. Por ello, definiremos primero esta operación. Para $0 \leq i < 2^m - 1$, X^i se divide entre $p(X)$, dando como resultado:

$$X^i = q_i(X)p(X) + a_i(X) \tag{A.15}$$

donde,

$$a(X) = a_{i0} + a_{i1}X + a_{i2}X^2 + \cdots + a_{i,m-1}X^{m-1} \tag{A.16}$$

a_i es de grado menor a m bajo $GF(2)$. Al ser X y $p(X)$ respectivamente primos, X^i no es divisible entre $p(X)$, por lo que para cualquier $i \geq 0$, tenemos:

$$a_i(X) \neq 0 \tag{A.17}$$

$$a_i(X) \neq a_j(X); 0 \ \leq \ i, \ j < 2^m - 1 \ y \ i \neq j \tag{A.18}$$

$$a_i(X) \neq 0 \tag{A.19}$$

$$a_i(X) \neq a_j(X); 0 \ \leq \ i, \ j < 2^m - 1 \ y \ i \neq j \tag{A.20}$$

De estas ecuaciones, obtenemos que, para $i = 0, 1, 2, \ldots, 2^{m-1}$ se obtienen 2^{m-1} polinomios $a_i(X)$ distintos del polinomios zero de grado $m-1$ o menor. Si reemplazamos X por α tenemos:

$$\alpha^i = a(\alpha) = a_{i0} + a_{i1}\alpha + a_{i2}\alpha^2 + \cdots + a_{i,m-1}\alpha^{m-1} \tag{A.21}$$

Este polinomio representa 2^{m-1} elementos distintos a $0, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{2^m-2}$. Hay 2^{m-1} polinomios distintos en α bajo $GF(2)$, los cuales representan 2^{m-1}elementos distintos de cero dentro del conjunto F.

La operación de adición " + " se define como:

$$0 + 0 = 0 \tag{A.22}$$

para $0 \leq i,\, j < 2^m - 1$,

$$\begin{aligned} &0 + \alpha^i = \alpha^i + 0 = \alpha^i \\ &\alpha^i + \alpha^j = (a_{i0} + a_{i1} + \cdots + a_{i,m-1}\alpha^{m-1}) + (a_{j0} + a_{j1}\alpha + \cdots + a_{j,m-1}\alpha^{m-1}) \\ &= (a_{i0} + a_{j0}) + (a_{i1} + a_{j1})\alpha + \cdots + (a_{i,m-1} + a_{j,m-1})\alpha^{m-1} \end{aligned} \tag{A.23}$$

donde la suma es módulo 2. Sería fácilmente demostrable que F es un grupo conmutativo respecto a la suma. Así, tenemos que los elementos del conjunto F son conmutativos respecto a la suma y que el conjunto de elementos distintos a 0 de F son también conmutativos respecto a la multiplicación. Por ello, tenemos que el conjunto

$$\mathbf{F} = \left\{0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^m - 2}\right\} \tag{A.24}$$

es un campo de Galois finito de 2^m elementos, $GF(2^m)$.

Ejemplo 8 *Sea $m = 3$, y $p_i(X) = 1 + X + X^3$, un polinomio primitivo bajo $GF(2)$. Al ser primitivo tenemos que $p_i(\alpha) = 1 + \alpha + \alpha^3 = 0$. Entonces, $\alpha^3 = 1 + \alpha$. Utilizando esta ecuación podemos construir $GF(2^3)$.*

$$\begin{aligned} &\alpha^3 = 1 + \alpha \\ &\alpha^4 = \alpha \ldots \alpha^3 = \alpha(1 + \alpha) = \alpha + \alpha^2 \\ &\alpha^{10} = \alpha^7 \ldots \alpha^3 = \alpha^3 \\ &\alpha^2 + \alpha^4 = \alpha^2 + \alpha + \alpha^2 = \alpha \end{aligned} \tag{A.25}$$

Representación exponencial	*Representación polinómica*	*Representación vectorial*
0	0	000
1	1	100
α	α	010
α^2	α^2	001
α^3	$1+\alpha$	110
α^4	$\alpha+\alpha^2$	011
α^5	$1+\alpha+\alpha^2$	111
α^6	$1+\alpha^2$	101

Cuadro A.1: *Elementos de* $GF(2^3)$ *generados por* $p_i(X) = 1 + X + X^3$

Representación exponencial	*Representación polinómica*	*Representación vectorial*
0	0	0000
1	1	1000
α	α	0100
α^2	α^2	0010
α^3	α^3	0001
α^4	$1+\alpha$	1100
α^5	$\alpha+\alpha^2$	0110
α^6	$\alpha^2+\alpha^3$	0011
α^7	$1+\alpha+\alpha^3$	1101
α^8	$1+\alpha^2$	1010
α^9	$\alpha+\alpha^3$	0101
α^{10}	$1+\alpha+\alpha^2$	1110
α^{11}	$\alpha+\alpha^2+\alpha^3$	0111
α^{12}	$1+\alpha+\alpha^2+\alpha^3$	1111
α^{13}	$1+\alpha^2+\alpha^3$	1001
α^{14}	$1+\alpha^3$	1001

Cuadro A.2: *Elementos de* $GF(2^4)$ *generados por* $p_i = 1 + X + X^4$

A.2. Propiedades de los campos extendidos de Galois $GF(2^m)$

Al igual que pasa a menudo en el álgebra ordinaria, donde una ecuación puede no tener raíces reales pero sí tenerlas complejas, en el caso de polinomios con coeficientes bajo $GF(2)$ pasa lo mismo, esto es, polinomios con coeficientes $GF(2)$ pueden no tener raíces en $GF(2)$ pero si en campos extendidos de $GF(2)$.

Por ejemplo, el polinomio irreducible $p_i(X) = 1 + X + X^4$ no tiene raíces en el campo $GF(2)$;, sin embargo, tiene 4 raíces en el campo extendido $GF(2^4)$.Así, se puede demostrar que $\alpha^7, \alpha^{11}\alpha^{13}$ y α^{14} son raíces del polinomio.

Teorema 5 *Sea $f(X)$ un polinomio con coeficientes del campo $GF(2)$. Sea β un elemento del campo extendido $GF(2^m)$. Si β es una raíz de $f(X)$, entonces para cualquier $l \geq 0$, β^{2^l} es también raíz de $f(X)$.*

Sabemos que:

$$\beta^{2^m-1} + 1 = 0 \tag{A.26}$$

Así, β es raíz del polinomio $X^{2m-1} + 1$. Cualquier elemento distinto de 0 del $GF(2^m)$ es raíz de $X^{2^m-1} + 1$. Al ser $2^m - 1$ el grado de $X^{2^m-1} + 1$, estos $2^m - 1$ forman todas las raíces de $X^{2^m-1} + 1$. Al ser también 0 la raíz del polinomio X, podemos afirmar que los elementos del campo $GF(2^m)$ generan todas las raíces de $X^{2^m-1} + 1$.

De este modo, tenemos que cualquier elemento β en $GF(2^m)$ es raíz de $X^{2^m} + X$. Puede ser que sea también raíz de un polinomio de grado menor de 2^m.

Definición 4 *El polinomio de grado mínimo $\Phi(X)$, cuyos elementos pertenecen a $GF(2)$, que tiene a β como raíz, se denomina polinomio mínimo de β.*

De *Lin−Cost* obtenemos los siguientes teoremas respecto a los polinomios mínimos:

Teorema 6 *El polinomio mínimo $\Phi(X)$ de un elemento del campo β es irreducible.*

Teorema 7 *Sea $f(X)$ un polinomio en $GF(2)$. Sea $\Phi(X)$ el polinomio mínimo de un elemento del campo β. Si β es raíz de $f(X)$, entonces $f(X)$ es divisible entre $\Phi(X)$.*

Teorema 8 *el polinomio mínimo $\Phi(X)$ de β en $GF(2^m)$ divide a $X^{2^m-1}+1$.*

Teorema 9 *Sea $f(X)$ un polinomio irreducible en $GF(2)$. Sea β un elemento en $GF(2^m)$. Sea $\Phi(X)$ el polinomio mínimo de β. Si $f(\beta) = 0$, entonces $\Phi(X) = f(X)$.*

Teorema 10 *Sea β un elemento en $GF(2^m)$, y $\boldsymbol{e}$ el menor entero no negativo tal que $\beta^{2^e} = \beta$ entonces,*

$$f(X) = \prod_{i=0}^{e-1}(X + \beta^{2^i}) \tag{A.27}$$

Teorema 11 *Sea $\Phi(X)$ el mínimo polinomio de un elemento β en $GF(2^m)$. Sea $\boldsymbol{e}$ el entero menor tal que $\beta^{2^e} = \beta$, entonces*

$$\Phi(X) = \prod_{i=0}^{e-1}(X + \beta^{2^i}) \tag{A.28}$$

Teorema 12 *Sea $\Phi(X)$ el mínimo polinomio de un elemento β en $GF(2^m)$. Sea $\boldsymbol{e}$ el grado de $\Phi(X)$. Se cumple que $\boldsymbol{e}$ es el entero menor tal que $\beta^{2^e} = \beta$, además $\boldsymbol{e} \geq m$.*

Teorema 13 *Si β es un elemento primitivo de $GF(2^m)$, todos sus conjugados, $\beta^2, \beta^4, \beta^8, \ldots$ son también elementos primitivos de $GF(2^m)$.*

Teorema 14 *Si β es un elemento de orden n en $GF(2^m)$, todos sus conjugados tienen el mismo orden n.*

Apéndice B

Decodificación de códigos BCH mediante el algoritmo de Euclides

Dados dos números A y B, el algoritmo de Euclides realiza una serie de combinaciones lineales hasta obtener el máximo común divisor $C = MCD(A, B)$. En este anexo se mostrará la utilización del algoritmo de Euclides para la decodificación de los códigos BCH.

B.1. Algoritmo de Euclides

Como hemos apuntado anteriormente, el algoritmo de Euclides permite obtener el máximo común divisor de dos números:

$$MCD(A, B) = C, \text{ entonces } A \; mod \; C = 0 \text{ y } B \; mod \; C = 0. \tag{B.1}$$

$$(A - K \cdot B) \; mod \; C = 0, K \in \mathbb{Z} \tag{B.2}$$

$$A \; mod \; B = \; M \rightarrow A = KB + M \rightarrow M = A - KB \tag{B.3}$$

Por lo tanto:

$$(A - KB)\ mod\ C = \ M\ mod\ C = 0 \tag{B.4}$$

Básicamente, el algoritmo de Euclides es ir dividiendo todos los restos que vayamos obteniendo, el último resto distinto de 0 es C.

Ejemplo 9 *MCD (84,49) = C*

El residuo es 35, dividimos éste por 49:

$49 \div 35 = 1 \times 35 + 14$

El residuo ahora es 14. Hacemos la operación consecutivamente hasta que hallemos un residuo 0:

$35 \div 14 = 2 \times 14 + 7$

$14 \div 7 = 2 \times 7 + 0 \leftarrow Ok.$

Como podemos ver, 7 es el último residuo, por lo que es el MCD.

B.2. Algoritmo extendido de Euclides

El algoritmo extendido de Euclides también nos sirve para hallar, mediante combinaciones lineales, dos números enteros o dos polinomios S y T tales que:

$$M.C.D(A, B) = C = SA + TB \tag{B.5}$$

Sea A y B dos números enteros tales que $A < B$ o, en forma polinómica, $deg\ (A) > deg\ (B)$. Partimos de las condiciones iniciales $r_{-1} = A$ y $r_0 = B$, los residuos de A y B, respectivamente. Ahora, al igual que hicimos en el algoritmo de Euclides, vamos obteniendo los residuos recursivamente. El valor de r_i se obtiene como residuo de la división de r_{i-2} entre r_{i-1} :

$$r_{i-2} = q_i r_{i-1} + r_i \tag{B.6}$$

donde $r_i < r_{i-1}$ o en el equivalente polinómico $deg\ (r_i) < deg\ (r_{i-1})$. La ecuación recursiva es pues:

$$r_i = r_{i-2} - g_i r_{i-1} \tag{B.7}$$

Las expresiones para s_i y t_i también se obtienen de

$$r_i = s_i A + t_i B \tag{B.8}$$

y los coeficientes de forma recursiva tienen la forma:

$$\begin{aligned} s_i &= s_2 - q_i s_{i-1} \\ t_i &= t_2 - q_i t_{i-1} \end{aligned} \tag{B.9}$$

Entonces,

$$\begin{aligned} r_{-1} &= A = (1)A + (0)B \\ r_0 &= B = (0)A + (1)B \end{aligned} \tag{B.10}$$

Las condiciones iniciales son:

$$s_{-1} = 1, t_{-1} = 0 \tag{B.11}$$

Ejemplo 10 *sea M.C.D (49,11) ¿ Cual es* 11^{-1} *en* Z_{49}*?*

$$\begin{aligned} 49 \cdot 11 + 11 \cdot 0 &= 49 \qquad (1) \\ 49 \cdot 0 + 11 \cdot 1 &= 11 \qquad (2) \end{aligned}$$

$49 \div 11 = 11 \cdot 4 + 5$, *residuo* $= 5$

$49 \cdot 1 - 4 \cdot 11 = 5$

combinamos linealmente (1) y (2)

$$\begin{aligned} &1 \cdot (1) - 4 \cdot (2) = 5 \\ &49 \cdot 1 + (-4) \cdot 11 = 5 \qquad (3) \end{aligned}$$

$11 \div 5 = 2 \cdot 5 + 1$, *residuo* $= 1$

$$\begin{aligned} &11 \cdot 1 - 2 \cdot 5 = 1 \\ &1 \cdot (2) - 2 \cdot (3) = 1 \end{aligned}$$

$$\begin{aligned} &11 = 49 \cdot 0 + 11 \cdot 1 \\ &\underline{(-2)(5 = 49 \cdot 1 + 11 \cdot (-4))} \\ &1 = 49 \cdot (-2) + 9 \cdot 11 \end{aligned}$$

$S = -2,\ T = 9$

también tenemos,

$11^{-1} = 9\ mod\ 49 = 9$

Es útil, a veces, colocar respectivos valores en una tabla:

i	$\mathbf{r}_i = \mathbf{r}_{i-2} - \mathbf{q}_i\mathbf{r}_{i-1}$	$\mathbf{q}_i$	$\mathbf{s}_i = \mathbf{s}_{i-2} - \mathbf{q}_i\mathbf{s}_{i-1}$	$\mathbf{t}_i = \mathbf{t}_{i-2} - \mathbf{q}_i\mathbf{t}_{i-1}$
-1	49	-	1	0
0	12	-	0	1
1	5	4	1	-4
2	1	2	-2	9

Cuadro B.1

Este algoritmo es muy útil para resolver la *ecuación clave*:

$$\sigma(X)S(X) = -W(X) + \mu(X)X^{2t} \tag{B.12}$$

lo que es lo mismo que decir

$$\{\sigma(X)S(X) + W(X)\} \mod(X^{2t}) = 0 \tag{B.13}$$

Recordemos que el polinomio evaluador de errores $\mathbf{W}(X)$ nos proporciona las posiciones de error e_{jl}, donde X^{2t} se correspondería a A y el polinomio de síndromes $\mathbf{S}(X)$ se correspondería a B.

Esta ecuación nos sirve para la decodificación de los códigos BCH.

Si ponemos la ecuación clave en forma recursiva tenemos:

$$r_i(X) = s_i(X)X^{2t} + t_i(X)S(X) \tag{B.14}$$

Si multiplicamos esta ecuación por la constante λ para forzar que el polinomio resultante sea mónico:

$$\lambda r_i(X) = \lambda s_i(X)X^{2t} + \lambda t_i(X)S(X) = -\mathbf{W}(X) = -\mu(X)X^{2t} + \sigma(X)S(X) \tag{B.15}$$

Así tenemos:

$$\begin{aligned} \mathbf{W}(X) &= -\lambda r_i(X) \\ \sigma(X) &= \lambda t_i(X) \end{aligned} \tag{B.16}$$

Ejemplo 11 *Sea el código* $C_{BCH}(15, 7)$ *con* $t = 2$. *Se recibe el polinomio* $\boldsymbol{r} =$ (100000001000000), *que en forma polinómica es* $r(X) = 1 + X^8$. *Decodificar*

$\boldsymbol{r}$ mediante el algoritmo de Euclides. El primer paso es el computo de los síndromes:

$$s_1(\alpha) = r(\alpha) = \alpha^2$$
$$s_1(\alpha) = r(\alpha^2) = \alpha^4$$
$$s_1(\alpha) = r(\alpha^3) = \alpha^7$$
$$s_1(\alpha) = r(\alpha^4) = \alpha^8$$

De modo que el síndrome en forma polinómica nos queda de la siguiente manera:

$$S(X) = \alpha^2 + \alpha^4 X + \alpha^7 X^2 + \alpha^8 X^3$$

Colocamos los valores del algoritmo extendido de Euclides en una tabla:

$$r_i(X) = s_i(X)X^{2t} + t_i(X)S(X)$$

$\mathbf{i}$	$\mathbf{r}_i = \mathbf{r}_{i-2} - \mathbf{q}_i\mathbf{r}_{i-1}$	$\mathbf{q}_i$	$\mathbf{s}_i = \mathbf{s}_{i-2} - \mathbf{q}_i\mathbf{s}_{i-1}$	$\mathbf{t}_i = \mathbf{t}_{i-2} - \mathbf{q}_i\mathbf{t}_{i-1}$
-1	$X^{2t} = X^4$	-	1	0
0	$S(X) = \alpha^2 + \alpha^4 X + \alpha^7 X^2 + \alpha^8 X^3$	-	0	1
1	$\alpha^8 + \alpha^{13} X + \alpha^4 X^2$	$\alpha^6 + \alpha^7 X$	1	$\alpha^6 + \alpha^7 X$
2	α^5	$\alpha^4 X + \alpha^8$	$\alpha^4 X + \alpha^8$	$\alpha^{11} X^2 + \alpha^5 X + \alpha^3$

Cuadro B.2

*Las operaciones se llevan a cabo según el campo extendido $GF(2^4)$. Al ser módulo dos las sumas se corresponden a las restas. El cálculo se detiene cuando el grado del residuo r_i es mayor que t_i.

Una vez hechos los cálculos del algoritmo de Euclides calculamos λ tal que t_i sea mónico. Esta λ es $\lambda = \alpha^4$. Así, ya estamos capacitados para encontrar los valores de $\boldsymbol{W}(X)$ y $\sigma(X)$.

$$\boldsymbol{W}(X) = -\lambda r_i(X) = \alpha^4 \cdot \alpha^5 = \alpha^9$$
$$\sigma(X) = \lambda t_i(X) = \alpha^4 \cdot (\alpha^1 1 X^2 + \alpha^5 X + \alpha^3) = X^2 + \alpha^9 X + \alpha^7$$

Una vez hallados estos polinomios, el siguiente paso es proceder a realizar lo que se denomina la búsqueda de Chien, que consiste en ir sustituyendo las

X en $\sigma(X)$ por $1, \alpha, \alpha, \ldots, \alpha^n$, siendo $n = 2^m - 1$, con el fin de hallar las raíces del polinomio localizador de errores. Así, si α^h es una raíz. Ésta se localiza en la posición r_{n-h}.

$$\sigma(\alpha^0) = \sigma(1) = 1 + \alpha^9 + \alpha^7 = 1 + \alpha + \alpha^3 + 1 + \alpha + \alpha^3 = 0$$
$$\sigma(\alpha^1) = \sigma(1) = \alpha^2 + \alpha^{10} + \alpha^7 = \alpha^2 + 1 + \alpha + \alpha^2 + 1 + \alpha + \alpha^3 = \alpha^3 \neq 0$$
$$\sigma(\alpha^7) = \sigma(1) = \alpha^{14} + \alpha^{16} + \alpha^7 = 1 + \alpha^3 + \alpha + 1 + \alpha + \alpha^3 = 0$$

Así la posición de los errores está en $r_{n-h} = r_{15} = r_0$, y en $r_{n-h} = r_8$. De modo que el polinomio de error es:

$$e(X) = 1 + X^8$$

No es necesario hallar aquí los valores de error, ya que al ser un código BCH binario, es seguro que es 1; más si fuera un código no binario, se hallarían a partir de la siguiente ecuación:

$$e_{jl} = \frac{W(\alpha^{-ji})}{\sigma'(\alpha^{-ji})}$$

donde:

$$\sigma'(X) = 2X + \alpha^9 = X + X + \alpha^9 = \alpha^9 \tag{B.17}$$

Al ser también,

$$\boldsymbol{W}(X) = -\lambda r_i(X) = \alpha^4 \cdot \alpha^5 = \alpha^9 \tag{B.18}$$

como era de esperar, por ser un código binario, obtenemos como resultado el valor de error constante 1:

$$e_{jl} = \frac{W(\alpha^{-ji})}{\sigma'(\alpha^{-ji})} = \frac{\alpha^9}{\alpha^9} = 1 \tag{B.19}$$

De este modo, continuando con el vector de error, para obtener el código original, se debe sumar el vector de error al código recibido:

$$m(X) = e(X) + r(X) = 1 + X^8 + 1 + X^8 = \boldsymbol{0} \tag{B.20}$$

De modo que la secuencia transmitida a través del canal es el vector nulo.

Apéndice C

Funciones relevantes implementadas en Matlab

C.1. Función cola.m

```
1   function bits=cola(trellis,estado)
2   %bits = cola(trellis)
3   %esta funcion te calcula los bits de cola
4   %que te llevan al estado 1 desde estado con menos
5   %iteraciones
6   %
7   INF=10000;
8   trellis.nextStates=trellis.nextStates+1;
9   estado=estado+1;
10  filas=trellis.numStates;
11  col=log2(filas);
12  nivel=1;
13  grafo=cell(filas,col+1);
14
15  nodo.prev=estado; %estado anterior del nodo
16  nodo.peso=INF;%peso acumulado inicialmente muy alto
17  nodo.hist=zeros(1,col);%historico de los caminos
18
19  nodo.activado=0;
20
21  for i=1:filas
22      for j=1:col+1
23          grafo{i,j}=nodo;
24      end
25  end
26
```

```
27    grafo{estado,1}.activado=1;
28    grafo{estado,1}.peso=1;
29    while (nivel<=col+1)
30
31  for k=1:filas
32    if (grafo{k,nivel}.activado==1)
33       estado0=trellis.nextStates(k,1);% que estados parten del mio
34       estado1=trellis.nextStates(k,2);
35
36       grafo{estado0,nivel+1}.activado=1;
37       predecesor=find(trellis.nextStates==estado0)';
38        %busco que estados preceden al que he llegado mediante un 0
39       predecesor(2)=predecesor(2)-trellis.numStates;
40       %compruebo que no hay otro camino mas corto activo
41       if(grafo{predecesor(1),nivel}.peso<grafo{predecesor(2),nivel}.peso)
42           grafo{estado0,nivel+1}.hist=grafo{predecesor(1),nivel}.hist;
43           grafo{estado0,nivel+1}.hist(nivel)=0;
44           grafo{estado0,nivel+1}.peso=grafo{predecesor(1),nivel}.peso+1;
45       else
46           grafo{estado0,nivel+1}.hist=grafo{predecesor(2),nivel}.hist;
47           grafo{estado0,nivel+1}.hist(nivel)=0;
48           grafo{estado0,nivel+1}.peso=grafo{predecesor(2),nivel}.peso+1;
49       end
50       grafo{estado1,nivel+1}.activado=1;
51       predecesor=find(trellis.nextStates==estado1)';
52        %busco que estados preceden al que he llegado mediante un 1
53       predecesor(2)=predecesor(2)-trellis.numStates;
54         %me quedo con el camino mas corto de los dos
55       if(grafo{predecesor(1),nivel}.peso<grafo{predecesor(2),nivel}.peso)
56           grafo{estado1,nivel+1}.hist=grafo{predecesor(1),nivel}.hist;
57           grafo{estado1,nivel+1}.hist(nivel)=1;
58           grafo{estado1,nivel+1}.peso=grafo{predecesor(1),nivel}.peso+1;
59       else
60           grafo{estado1,nivel+1}.hist=grafo{predecesor(2),nivel}.hist;
61           grafo{estado1,nivel+1}.hist(nivel)=1;
62           grafo{estado1,nivel+1}.peso=grafo{predecesor(2),nivel}.peso+1;
63       end
64
65
66
67    end
68
69  end
```

```
71
72      if(grafo{1,nivel+1}.activado==1)
73       %esto quiere decir que he llegado al estado 0
74          bits=grafo{1,nivel+1}.hist;
75          nivel=col+2;%para terminar el bucle
76      end
77
78          nivel=nivel+1;
79
80  end
```

C.2. Función gencodi.m

```
1  function [secuencia códigos longitudes blancos]=gencodi(num,...
2            trellis1,trellis2,trellis3,inter1,inter2,inter3)
3  % Funcion que genera num secuencias codigo, es decir
4  %claves de 512 bits, y sus respectivas secuencias codificadas
5
6  secuencia=zeros(num,512);
7
8  for i=1:num
9      secuencia(i,:)=round(rand(1,512));
10 end
11
12 long1=log2(trellis1.numStates)*2; %longitud bits de cola *2 ...
13                                    debido a los bits secuenciales
14 códigos1=zeros(num,512*3+long1*3);
15 for i=1:num
16     msg=secuencia(i,:);
17     x=turbocod(msg,trellis1,inter1);
18     códigos1(i,:)=x;
19 end
```

```
long2=log2(trellis2.numStates)*2; %longitud bits de cola *2 ...
                                   debido a los bits secuenciales
códigos2=zeros(num,512*3+long2*3);
for i=1:num
    msg=secuencia(i,:);
    x=turbocod(msg,trellis2,inter2);
    códigos2(i,:)=x;
end
long3=log2(trellis3.numStates)*2; %longitud bits de cola *2 ...
                                   debido a los bits secuencialess
códigos3=zeros(num,512*3+long3*3);
for i=1:num
    msg=secuencia(i,:);
    x=turbocod(msg,trellis3,inter3);
    códigos3(i,:)=x;
end

longitudes=[length(códigos1(1,:)) length(códigos2(1,:))...
                                  length(códigos3(1,:))]; % esto es
                                  porque las longitudes no son iguales
                                  a causa de los distintos trellis
códigos=horzcat(códigos1,códigos2,códigos3);
[códigos blancos]=aleatorizador(códigos);
secuencia=horzcat(secuencia,secuencia,secuencia);
```

C.3. Función comfabular.m

Esta función realiza la confabulación de las marcas. Hay dos modalidades; por bloques de tres bits o bien bit a bit. Como se puede comprobar, la opción de bloque de tres bits está comentada en la función, por lo que con la configuración mostrada trabajaría en una confabulación bit a bit. Se han truncado los casos con 6 y 10 confabuladores por temas de espacio.

```
1    function [comfabulats malos]=comfabular(codis,num,atac)
2    %function comfabulats=comfabulacio(codis,num,atac)
3    %
4    % Esta funcion te genera una confabulacion
      % de de num comfabuladores
5    % de entre todos los posibles códigos codis.
6    % Num tiene que ser par.
7    % número de confabuladores
8    [filas,columnas]=size(codis);
9    malos=ceil(filas*rand(1,num));
10   comfabulats=zeros(num/2,columnas);
11   tmp=[];
12   if(atac==5)
13       final=columnas;
14       %final=columnas/3;
15   switch num;
16       case 2
17           for i=1:final
18           cual=ceil(2*rand(1));
19           if (cual==1)
20     %malo=conf_atac(codis(malos(1),:),codis(malos(2),:),atac);
21               aux=codis(malos(1),i);
22           else
23               aux=codis(malos(2),i);
24           end
25           tmp=horzcat(tmp,aux);
26           end
```

```
27       case 4
28            for i=1:final
29               cual=ceil(4*rand(1));
30               if (cual==1)
31                   %aux=codis(malos(1),(i-1)*3 +1:i*3);
32
33                   aux=codis(malos(1),i);
34               elseif (cual==2)
35                   %aux=codis(malos(2),(i-1)*3 +1:i*3);
36
37                   aux=codis(malos(2),i);
38               elseif(cual==3)
39                   %aux=codis(malos(3),(i-1)*3 +1:i*3);
40
41                   aux=codis(malos(3),i);
42               elseif(cual==4)
43                   %aux=codis(malos(4),(i-1)*3 +1:i*3);
44
45                   aux=codis(malos(4),i);
46               else
47                   disp('se ha producido un error desconocido')
48               end
49                tmp=horzcat(tmp,aux);
50
51            end
52
53       case 6
            (...)
79       case 10
             (...)
119  end
120  malo=tmp;
121  else
122  malo=conf_atac(codis(malos(1),:),codis(malos(2),:),atac);
123  for i=2:num-1
124  malo=conf_atac(malo,codis(malos(i+1),:),atac);
125  end
126  end
127     comfabulats=malo;
```

C.4. Función testfinal.m

En esta función, cabe destacar que la opción elegida para la obtención de los usuarios deshonestos ha sido la decodificación turbo, quedándonos con los valores soft. Una vez obtenida la secuencia decodificada, se realiza una correlación cruzada de modo que los valores que superen un cierto umbral en sus máximos serán considerados deshonestos al ser las secuencias de mayor parecido. Las variables de esta función son los valores Lc, el trellis utilizado, el número de confabuladores, el interleaver de los RSC, el número de iteraciones, el tipo de confabulación y el umbral de decisión.

```
1   function maximos=testfinal5
2   %Funcion para obtener los usuarios deshonestos
3
4   %1000 usuarios ,6 deshonestos, umbral 30, Lc=0.75 ataque 1, 15
5   %iteraciones,trellis 6
6   global resultados;
7   global posicion;
8      inter=getinterleaver(2);
9      %Aquí introducir el número de códigos existentes
10      numclaves=1000;
11     %Aqui introducir el trellis
12     trellis=gettrellis(6);
13     %Aqui introducir el número de confabuladores
14     confabuladores=4;
15     % el siguiente valor es el de la constante de canal Lc
16     Lc=0.75;
17     %El siguiente valor es el número de iteraciones;
18     it=15;
19     %el siguiente valor es el umbral
20     umbral=30;
21     [buenos buenoscodif blancos]=gencodi2(numclaves,trellis,inter);
22    %cambiamos buenoscodif por blancos para comprobar
23     %que tal sin pasar porel decodificador.
24      [malo deshonestos]=comfabular(buenoscodif,confabuladores,5);
25     %para caso de no confabulación
26     %[malo deshonestos]=comfabular(buenos,confabuladores,5);
```

```
31
32      %decodificamos
33      %esto lo quitamos a posta en este caso no tenemos codificación.
34      x=decturbo(malo,Lc,trellis,inter,it);
35      %x=malo;
36      x_hard=(sign(x)+1)/2;
37      % realizamos una correlación para pillar los deshonestos
38      contador=1;
39      maximo=0;
40      pillados=zeros(1,30);
41      for j=1:numclaves
42
43           correlacion=xcorr(x,buenos(j,:));
44           maximo=correlacion(length(x));
45           %maximo=max(correlacion);
46           maximos(j)=maximo;
47
48           if(maximo>umbral)
49               pillados(contador)=j;
50               contador=contador+1;
51
52               end
53            end
54
55      aux=find(pillados>0);
56           nume=numel(aux)
57           resultados(1,posicion)=nume;
58           totalencontrados=0;
59           for k=1:nume
60      encontrado=numel( find(deshonestos==pillados(k)));
61      if(encontrado>0)
62          totalencontrados=totalencontrados+1;
63      end
64           end
65           falsospositivos=nume-totalencontrados;
66  resultados(2,posicion)=falsospositivos;
67  resultados(3,posicion)=totalencontrados;
68  deshonestos
69  end
```

C.5. Función testhard.m

Esta función ha sido creada para comprobar el número de bits que se corrigen mediante la codificación turbo. Por ello, es preciso quedarnos con los valores *hard* (unos y ceros) de la decodificación, a diferencia de lo que hacíamos en la función anterior, en la cual utilizábamos los valores *soft*, es decir con decimales. Una vez realizada la decodificación, realizamos una comparación restando la marca decodificada al resto de marcas del grupo.

```
1   function [result  correlacion resulthon ...
          resultmaloenA buenos buenosb]=test_hard
2   %Comparamos la capacidad de corrección de la codificación turbo.
3   %
4   %honin es el resltado de comparar un usario honesto con el grupo
5   %
6   global deshon;
7   global deshonestos;
8   %1000 usuarios ,6 deshonestos, umbral 30, Lc=1 ataque 1, 15
9   %iteraciones,trellis 4
10
11  inter=getinterleaver(2);
12  %Aquí introducir el número de códigos existentes
13  numclaves=1000;
14  %Aqui introducir el trellis
15  trellis=gettrellis(4);
16  %Aqui introducir el número de confabuladores
17  confabuladores=10;
18  %El siguiente valor es el número de iteraciones;
19  itb=5; %iteraciones del decodificador para decodificar honestos
20  [buenos buenoscodif blancos]=gencodi2(numclaves,trellis,inter);
21  [buenosb buenoscodifb blancosb]=gencodi2(numclaves,trellis,inter);
22  x_honcod=buenoscodifb(100,:); %usuario honesto codificado,
23                                    %perteneciente a la muestra A.
24  x_honcoda=buenoscodif(100,:);
25  %cambiamos buenoscodif por blancos para
26  %comprobar que tal sin pasar porel decodificador.
27  [malo deshonestos]=comfabular(buenoscodif,confabuladores,5);
28  malo2=comfabular2(buenos,confabuladores,5,deshonestos);
```

```
29      %[malo deshonestos]=comfabular(buenos,confabuladores,5);
30     %decodificamos
31     %esto lo quitamos a posta en este caso no tenemos codificación.
32     %x=malo;
33     % realizamos una correlación para pillar los deshonestos
34     contador=1;
35     maximo=0;
36
37    for r=1:5
38        switch r
39            case 1
40                Lc=0.75;
41                umbral=30;
42                it=30;
43                x=decturbo(malo,Lc,trellis,inter,it);
44                x_hard=(sign(x)+1)/2;
45                x_hondec=decturbo(x_honcod,Lc,trellis,inter,itb);
46                x_hondec=(sign(x_hondec)+1)/2;
47
48            case 2
49                Lc=0.75;
50                umbral=30;
51                it=60;
52                x=decturbo(malo,Lc,trellis,inter,it);
53                x_hard=(sign(x)+1)/2;
54                x_hondec=decturbo(x_honcod,Lc,trellis,inter,itb);
55                x_hondec=(sign(x_hondec)+1)/2;
56            case 3
57                Lc=1;
58                umbral=30;
59                it=30;
60                x=decturbo(malo,Lc,trellis,inter,it);
61                x_hard=(sign(x)+1)/2;
62                x_hondec=decturbo(x_honcod,Lc,trellis,inter,itb);
63                x_hondec=(sign(x_hondec)+1)/2;
64            case 4
65                Lc=1;
66                umbral=30;
67                it=60;
68                x=decturbo(malo,Lc,trellis,inter,it);
69                x_hard=(sign(x)+1)/2;
70                x_hondec=decturbo(x_honcod,Lc,trellis,inter,itb);
71                x_hondec=(sign(x_hondec)+1)/2;
72            case 5
73                x_hard=malo2;
74        end
```

```
75
76     pillados=0;
77     for j=1:numclaves
78
79          correlacion(r,j,:)=xcorr(x,buenos(j,:));
80          result(r,j)=numel(nonzeros(buenos(j,:)-x_hard));
81          resulthon(r,j)=numel(nonzeros(buenos(j,:)-x_hondec));
82          resultmaloenA(r,j)=numel(nonzeros(buenosb(j,:)-x_hard));
83     end
84     aux=find(pillados>0);
85  nume=numel(aux)
86  totalencontrados=0;
87  for k=1:nume
88     encontrado=numel( find(deshonestos==pillados(k)));
89     if(encontrado>0)
90         totalencontrados=totalencontrados+1;
91     end
92  end
93  falsospositivos=nume-totalencontrados
94  pillados=pillados(1:nume)
95    end
96  deshonestos
```

Bibliografía

[1] Shu Lin, Daniel J. Costello *Error Control Coding (2nd Edition)*, Prentice-Hall Inc., 2004.

[2] J. Castiñeira Moreira, P. Guy Farrell *Essentials of error-control coding*, John Wiley & Sons Ltd, 2006.

[3] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara "Soft-Output Decoding Algorithms in Iterative Decoding of Turbo Codes", pp. 63-87, 1996.

[4] S. Benedetto, G. Montorsi,"Unveiling Turbo Codes: Some Results on Parallel Concatenated Coding Schemes", *IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 42, NO. 2, MARCH 1996*, pp. 409-428, 1996.

[5] Mark Bingeman, Amir K. Khandani,"Symbol-Based Turbo Codes", *IEEE COMMUNICATIONS LETTERS, VOL. 3, NO. 10, OCTOBER 1999*, pp. 285-287, 1999.

[6] D. Divsalar, S.Dolinar, F. Pollara,"Iterative Turbo Decoder Analysis Based on Density Evolution", *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS, VOL. 19, NO. 5, MAY 2001*, pp. 891-907, 2001.

[7] Ricardo A. Podestá,"Códigos Cíclicos", *Jornadas de Criptografía y Códigos Autocorrectores (JCCA),20 al 24 de Noviembre 2006, Mar del Plata, Argentina.*, 2006.

[8] Joan Tomàs-Buliart, Marcel Fernandez, Miguel Soriano,"Improvement of Collusion Secure Convolutional Fingerprinting Information Codes", *Department of Telematics Engineering. Universitat Polit'ecnica de Catalunya.*

[9] D. Divsalar, F. Pollara,"Turbo Trellis Coded Modulation with Iterative Decoding for Mobile Satellite Communications".

[10] D. Divsalar, F. Pollara,"Multiple Turbo Codes for Deep-Space Communications", 1995.

[11] Marcin Sikora, Daniel J. Costello,"A New SISO Algorithm with Application to Turbo Equalization", *Department of Electrical Engineering, University of Notre Dame*, 2005.

[12] Charan Langton,"Turbo Coding and MAP Decoding - Example", 2006.

[13] "ETSETB. MATLAB. Fundamentos y/o Aplicaciones. Curso 06/07".

[14] Bernard Sklar,"Fundamentals of Turbo Codes".

[15] Richard Chávez, Gabriel Gamarra, Mauricio Pardo `http://www.tac-global.com`

[16] `http://physionet.cps.unizar.es/~eduardo/docencia/tvoz/rah/sld028.htm`

[17] Bernard Sklar, *Fundamentals of Turbo Codes* `http://physionet.cps.unizar.es/~eduardo/docencia/tvoz/rah/sld028.htm`

[18] `http://spanish.peopledaily.com.cn/31617/6832010.html`

[19] `http://es.globedia.com/pirateria-software-espana-nivel`

[20] `http://www.elmundo.es/elmundo/2009/05/06/navegante/1241603666.html`

[21] `http://es.wikipedia.org/wiki/Propiedad_intelectual`

[22] `http://en.wikipedia.org/wiki/Peer-to-peer`